DOCTORAL THESIS

Enabling Workspace Transference A cross-environment framework for application and data portability

Carter, Matthew

Award date:
2012

Enabling Workspace Transference

A cross-environment framework for application and data portability

Matt Carter BSc, MIT

Thesis submitted for the degree of Doctor of Philosophy to the School of Information Technology, Bond University, Australia

February 2012

Table of Contents

Abstract **6**

Statement of Original Authorship **7**

1 Introduction **1**
1.1 Motivation and statement of problems 1
1.2 Research question 2
1.3 Research deliverables 3

2 Literature Review **5**
2.1 History of software portability 5
2.2 Existing workspace transference methodologies 6
2.2.1 Application podding 7
2.2.2 Application stasis 8
2.2.3 Virtualisation 10
2.2.4 Teleportation 13
2.2.5 System emulation 14
2.2.6 System boot 15
2.2.7 Web-based systems 16
2.2.8 Summary of workspace transference methodologies 17
2.3 Other portability issues 17
2.3.1 File synchronisation 18
2.3.2 Security 19
2.4 Literature Summary 19

3 Workspace Transference Framework **22**
3.1 Framework aims 22
3.1.1 Application portability 23
3.1.2 Data portability 25
3.1.3 Data sharing 26
3.1.4 Environmental interaction 27
3.1.5 Interface requirements 28
3.2 Portability infrastructure 29
3.2.1 Application portability 29
3.2.2 Data portability & sharing 31
3.2.3 Environmental Settings 31
3.2.4 Interface Requirements 32
3.3 Framework Summary 33

4 Evaluation Criteria **34**
4.1 Persona use cases 34
4.1.1 Novice 36
4.1.2 Knowledge Worker 37
4.1.3 Developer 39
4.1.4 Rejected personas 40
4.2 Technical criteria 40
4.2.1 Distribution 40
4.2.2 Synchronisation 42
4.2.3 Package management 43
4.2.4 Error handling 44
4.2.5 Development environment 46
4.3 Rejected evaluation criteria 47
4.4 Evaluation Criteria Summary 48

5 Prototype Design **50**

5.1 Framework integration 50
5.1.1 Application portability 50
5.1.2 Data portability & sharing 51
5.1.3 Environmental interaction 53
5.1.4 Interface requirements 55
5.1.5 Framework integration summary 55
5.2 Logical Architecture 56
5.2.1 Client-Side Components 57
5.2.2 Server-Side Components 62
5.3 Component interaction 63
5.3.1 Prototype installation 64
5.3.2 Synchronisation 64
5.3.3 Package installation 65
5.3.4 Package upgrade 65
5.3.5 Package removal 65
5.4 Prototype design summary 66

6 Initial Prototype 67
6.1 Technology selection 67
6.2 Evaluation methodology identification 69
6.3 Framework compliance 71
6.3.1 Application portability 71
6.3.2 Data portability & sharing 72
6.3.3 Environmental interaction 73
6.3.4 UI requirements 73
6.3.5 Framework compliance summary 74
6.4 Persona use cases 75
6.4.1 Novice Persona 75
6.4.2 Knowledge Worker Persona 82
6.4.3 Developer Persona 87
6.4.4 Persona use case summary 92
6.5 Technical criteria 95
6.5.1 Distribution 95
6.5.2 Synchronisation 96
6.5.3 Package management 97
6.5.4 Error handling 98
6.5.5 Development environment 98
6.6 Prototype evaluation and weaknesses 100
6.6.1 Prototype portability 102
6.6.2 Slow bootstrap 103
6.6.3 Community support 103
6.6.4 GUI integration 103
6.6.5 Maintenance 104
6.6.6 Revised action plan 105

7 Revised Prototype 106
7.1 Framework compliance 106
7.1.1 Application portability 106
7.1.2 Data portability & sharing 108
7.1.3 Environmental Interaction 110
7.1.4 Interface requirements 112
7.1.5 Framework compliance summary 114
7.2 Persona use cases 114
7.2.1 Bootstrap time 115
7.2.2 PyPredict 116
7.2.3 Web interface 116
7.2.4 Exception handling 117
7.2.5 Persona use case summary 117

7.3 Technical criteria 117
7.3.1 Distribution 117
7.3.2 Synchronisation 118
7.3.3 Package management 120
7.3.4 Error handling 121
7.3.5 Development environment 122
7.4 Prototype Evaluation 123

8 Conclusion 126
8.1 Thesis summary 126
8.2 Future research 127

A Glossary of terms 129

B Logic Model 132

References 161

List of figures

Figure 1. The workspace transference framework seperated as conceptual layers 30

Figure 2. Peer-to-peer prototype networking model 52

Figure 3. Network layout of centrally connected connected prototype nodes 52

Figure 4. Logical overview diagram of the prototype architecture 57

Figure 5. Emulation logic model 60

Figure 6. The prototype installation sequence presented in its initial form 76

Figure 7. The prototype installation begins 77

Figure 8. Prototype installation is completed 77

Figure 9. The prototype package installation screen 78

Figure 10. Installation status of a specific package 78

Figure 11. The word processor installation of is complete 79

Figure 12. System tray icons displaying a installed software 80

Figure 13. Unsaved work in the OpenOffice word processor 80

Figure 14. The same document now with purposely overwritten text 81

Figure 15. Extended installation screen showing more options 83

Figure 16. The now installed Firefox package is updated in the package manager window 84

Figure 17. The installed Firefox package's main window 85

Figure 18. The same web-browsing session now installed on a Microsoft Windows machine 87

Figure 19. The result of a simple Perl program being compiled on an Ubuntu Linux system 89

Figure 20. The same application running under the prototype's portability environment 90

Figure 21. Installing the prototype within a purely console-driven environment 91

Figure 22. The package creation process as seen in the console 92

Figure 23. Symbols used in this chapter to denote the logical flow of execution 133

Figure 24. Overall conceptual logical flow diagram of the core system 134

Figure 25. Logical flow diagram of the GUI sub-system 139

Figure 26. Logical flow diagram of the watcher sub-system 142

Figure 27. Logical flow diagram of the synchroniser sub-system 144

Figure 28. Logical flow diagram of the Unison sub-system 146

Abstract

Workspace transference is defined as the transportation of applications and associated data between different computer environments regardless of hardware, operating systems or networks. Existing research only provides partial solutions to workspace transference. A gap still exists to unify this research under a set of requirements. Therefore there is a need for an integrated framework that binds specific requirements together to provide a complete solution. This thesis defines the concept of workspace transference and constructs a framework which provides a solution to this research problem.

A series of evaluation criteria will then be created to assess any constructed solution followed by a prototyping methodology to construct a series of prototypes to test against the workspace transference framework. Two prototype implementation cycles are described using the Perl and Python development environments. Each prototype is tested against a series of technical criteria derived from the workspace transference framework and a use case task list generated for a range of user personas. The final prototype performed all the use case tasks well and met the stringent technical criteria. This outcome indicates that a comprehensive solution to workspace transference is a feasible and practical proposition.

The workspace transference framework unifies the existing practical solutions for workspace transference and provides a technological yardstick against which future solutions can be judged.

The final version of the prototype named WorkTran meets all the requirements of the workspace transference framework. WorkTran supports three major operating systems on Windows, Macintosh OSX and several varieties of Linux, and forms a significant research outcome.

Statement of Original Authorship

The work contained in this thesis has not been previously submitted for a degree or diploma at any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Signature:

Date:

Chapter 1. Introduction

1.1 Motivation and statement of problems

The use of multiple personal computers (PCs) is becoming increasingly widespread. It is now not uncommon for a person to own more than one PC, each of which can contain different file sets, applications, settings and locally attached hardware. These machines are effectively stand-alone workstations that function independently of one another. Substantial effort and technical expertise is needed to allow these multiple machines to share resources within a common work environment.

One solution is the creation of a personal home network that typically takes the form of a number of computers sharing resources amongst other workstations using a personal peer-to-peer file system. A limitation with this solution is that in order to share resources, each node must maintain physical connectivity to the network and any interruption to this required constant connectivity effectively cuts that peer machine off from the rest of the group. This problem is compounded further in that most applications are incapable of being shared between machines and require full and separate installation on each computer.

In order to explore the problem area, it is important to first determine terminology. The term ‘workspace’ is used in this research to define a set of applications and files in use by a user on one or more machines. A workspace typically consists of a combination of programs or utilities along with any personal files owned by the user. A workspace can be present on any number of machines, from home desktops or laptops to personalised computers within a company. In every one of these possible workspace scenarios, a user typically utilises a number of common utilities such as word processors, web browsers, email clients or other suites of programs based on their own personal preferences or institutional policies.

Workspace transference, the concept defined in this research, enables a workspace to be transported and replicated across multiple machines. Before the inception of workspace transference, a user was required to manually set up each environment, install preferred applications, and adjust settings and other configurations to recreate the same workspace. Workspace transference would automatically provide portability as the user physically moves from machine to machine.

The currently accepted method of transporting a workspace to other environments is to move the user's personal files and application library on a portable media device such as a Universal Serial Bus (USB) memory stick, a partial solution to the data portability problem. These media storage-based methods are potential single points of failure that can lead to unexpected data loss. Data loss can occur because of a media device being physically damaged, data corruption or user-invoked loss such as the accidental deletion of data. Regular backups are frequently promoted as a method to eliminate this risk, but habitual and well-maintained backups are rarely performed by most users.

A recent innovation in the absence of workspace transference is the concept of cloud-connected desktops[1]. In its most basic form, cloud-based desktop systems present a graphical or shell-based interface forwarded from the remote machine. Protocols such as Microsoft's Remote Desktop

Protocol[2] or virtual network computing (VNC)[3] allow for separation of the processing back end (hosted in the cloud) and the presentation hardware (situated physically with the user). Cloud-based systems can be thought of as terminal devices where, instead of having to maintain the computer himself, the user would simply connect to a desktop interface maintained elsewhere on dedicated and scalable server hardware. In this way, the cloud-hosted environment effectively acts like a television screen carrying the output of the actual processing component of the working server, which can cost-effectively and efficiently manage many of these interactions simultaneously. These systems are maintained centrally and allow the user to perform daily operations without the need for regular maintenance, patches, upgrading or other activities that can be conducted remotely by a technical professional.

The cloud-based model is an offshoot of the network computer (NC)[4] promoted by Sun Microsystems and Oracle, and both companies provided heavy support with what was once seen as a strong competitor of the fat-client–based PC. Despite these efforts, the original NC concept was not widely adopted for reasons such as the enforced closed standards that these NC systems operated on and, most importantly, the ever-decreasing cost of PCs and laptop computers compared to the increasingly expensive prices of lesser NC hardware. The adoption rate for NC-based hardware is currently extremely limited and has been largely surpassed by fat-client PCs.

Portability at the application level has been addressed by projects such as PortableApps[5] or MojoPac[6], which provide stand-alone binary installation that can be transported on removable media. While some larger applications such as the Firefox web browser and the Thunderbird mail client are supported within the PortableApps library, these applications suffer from the inability to adjust to the current host machine configuration. Network connection settings such as proxy server details are not automatically determined by the applications and require manual user intervention to supply this information to each machine. Cross-platform capabilities, data security and sharing solutions are also lacking. Likewise, MojoPac suffers from its Windows-only nature, which makes it unavailable to users of alternative operating systems (OSs).

1.2 Research question

Recent studies have shown that the average family of four in western industrialised countries has access to at least two ‘smart’ devices within the home[7]. This access to computers features the convenience of user mobility between these devices. Assuming an average technically-enabled worker has a home computer, mobile phone and workplace computer, each individual has access to at least three computers. Exposure to various computers increases further if the workplace does not issue fixed offices or the worker’s office location changes often.

Due to these scenarios, it is possible for a worker to have to maintain several workplaces simultaneously. A home machine may contain various email, web browser and entertainment software. A workplace machine could contain an office suite, local business software and files. A mobile workplace such as a laptop could contain a mixture of the two environments. Likewise, smaller niche devices such as mobile phones or external or USB drives (for use in other machines) could contain similar workplace software suites. The complexity of moving between these

increasingly varied workspaces is compounded further by the varied environments each of these machines exhibits. The home machine may run Mac OS X, the workplace machine may run Windows 7 (64-bit) and the laptop may run Windows 7 (32-bit). Each of these environments has different software or hardware, yet all are expected to contain the same intrinsic tools the worker requires.

As a literature review has shown, some niche solutions exist to bridge certain gaps in software mobility, but none successfully address all of the workspace transference issues. As the review summarised in Table 1 indicates, each existing technical or research solution is limited to its own operative area and fails to cover all of the requirements that would enable a casual user the requisite portability.

The research question for this paper can be stated as:

How can one best create an operational workspace transference environment that is free of platform limitations and restrictions to support a typical suite of programs of any origin and benefit most users?

1.3 Research deliverables

While some existing literature does cover certain aspects of workspace transference, a fully comprehensive solution to the problem is lacking. The overall aim of this research is to provide a comprehensive workspace transference solution, particularly within the areas of data accessibility and application portability. This goal may be achieved through the construction of a workspace transference framework that defines a complete set of requirements for a potential solution. This framework represents the first definition and unified study of workspace transference. The study begins with an investigation of existing research to find common ground and identify deficiencies within the current workspace transference marketplace. Once the workspace transference framework has been defined, a series of prototypes will be constructed to demonstrate the practicality and effectiveness of the framework in real-world situations.

As stated above, the research question for this paper can be stated as:

How can one best create an operational workspace transference environment that is free of platform limitations and restrictions to support a typical suite of programs of any origin and benefit most users?

The following thesis deliverables are derived from the above research question:

1. **Framework specification:** The specification of a workspace transference framework that involves the creation of an environment allowing for portability of software prototypes between various hardware and software platforms.
2. **Prototype design and construction:** The design and construction of a prototype using a cyclic refinement model that improves it at each iteration. The design and implementation process is assessed at the end of each construction phase with to the goal improving the overall implementation in each cycle.

3. **Prototype evaluation:** An assessment of the prototype developed in the above deliverable. Should multiple design and construction stages be required, this stage will also be repeated. Technical criteria are used to assess solution execution towards moving a set list of software packages representative of a typical software suite between hardware and software platforms.
4. **Use case design and evaluation:** Extending the above evaluation, the prototype will also be analysed against several case studies to demonstrate the potential application and benefits of the above framework.

Using prototyping methodology, a workable solution (known as WorkTran) will be created and undergo a series of refinements and evolutions against the workspace transference framework. This cyclic refinement will continue until a defined stop condition has been reached. Upon reaching the stop condition, the prototype will be seen as having acceptable implementation and conforming to the workspace transference framework.

After this implementation stage, a series of personas is constructed and applied to each prototype in order to assess its success. These personas represent a cross-section of the user base with the intention of providing a suitable and stable prototype matching the needs of all personas. Each of the prototypes is assessed against a series of use case tasks constructed for these personas. In addition to these assessments, a number of technical criteria are generated to measure the prototype success from both user and technical implementation perspectives. These technical criteria address the functionality and suitability of each prototype's programming language, the capability of cross-platform programming, graphical user interface (GUI) compatibility and other technical factors affecting the successful implementation of the cross-platform prototype development. Testing the prototype this way uncovers the deficiencies inherent to each implementation cycle in order to refine and perfect each successive cyclic prototype. At the end of the evaluation process, the final prototype should meet the requirements of the workspace transference framework.

The core component of this research is the formation of a framework that demonstrates the workspace transference concept as well as a prototype demonstrating its feasibility. The constructed workspace transference framework can be used by future researchers to verify that their solutions provide comprehensive workspace transference.

In the next chapter, a literature review will be presented that assesses the existing workspace transference research.

Chapter 2. Literature Review

Portability was defined as 'the ease with which a program can be transferred from one environment to another' by Poole and Waite in the seminal paper *Portability and Adaptability*[8]. In this work, Poole and Waite defined three factors that influence portability: machine hardware, OS and programming language. The paper itself was primarily concerned with the portability of a programming language to other platforms and furthered the concept of abstract machine computation[9]. The actual concept of moving a program regardless of the programming language has been uncovered only recently with the modern concept of the proprietary pre-compiled application.

Workspace transference attempts to take the Poole portability definition and extend it to include any data or application accessed on multiple machines by a single user. Workspace transference should be the goal regardless of whether the machine is personal property, a shared workplace asset or a temporary use or one-off use machine.

In this chapter, the history of portability is reviewed to study the underpinning practice in the area of computer science. Existing research and commercial projects are surveyed to show the current level of portability within the computing industry.

2.1 History of software portability

The concept of portability between distinct machine architectures has formed the basis of some of the larger technology companies in existence today. Microsoft itself owes its own founding to the successful migration of the BASIC interpreter to the Altair platform[10]. Early implementations of the seminal C language quickly spawned many portability efforts that are largely responsible for the widespread use of the language. However, these efforts are largely dependent on two primary factors: 1) the availability of open access to the application's source code and 2) the need to re-compile the application for use by each platform.

In the early days of the computing revolution, neither of these two factors represented any significant obstacle. Source code was almost always distributed rather than buried in a platform's proprietary pre-compiled binary format. This is largely because the early hardware platforms supplied their own C (or other language) compilers to allow the software to run on the host hardware. The source code provided the portability and the compilers performed both the hardware individualisation and the platform binding.

In more recent times, the distribution of application source code remains prevalent among open communities such as various Free and Open Source Software (FOSS) projects, which keep the practice alive. Modern business practices representing the larger software companies, however, ensure that their software, along with their corporate secrets and business intelligence, remain proprietary binary applications. This distribution method is more efficient (as the application is already compiled and optimised), more secure for these businesses (as the business can compile and keep the inner workings of these projects safe from competitors) and easier for the user (since there

is no need to compile the source code). However, it also represents an obstacle for non-supported OSs and hardware platforms.

The most useful resources with regard to the portability concept are the experiences of IBM and the Computer Science Department of North Carolina State University, which have attempted to produce a portable C compiler[11] that can translate any source code into a local machine's optimised binary format.

When an application is not compiled specifically for a hardware platform but portability is required, a middleware layer (usually a programming or scripting language) is deployed to translate the higher-level instructions to the lower-level equivalents. One of the earlier examples of this was the P-Stat[12] statistical analysis program written in the FORTRAN interpreted programming language. In the early FORTRAN days, the language was not entirely cross-platform, so the P-Stat software used an overlay system that selected the correct functionality depending on the current host hardware and environment. In this system, P-Stat could maintain cross-platform compatibility to allow applications developed under it to be transported successfully between platforms, provided that the P-Stat system had existing compatible overlays that could be used within the active hardware/software combination.

As an early precursor to modern scripting languages, Pascal could compile its source code into P-code[13], a machine-independent pseudo-code format providing application portability onto any machine with a P-Machine virtual state machine. While practical, the P-Code system was exceptionally limited in areas such as graphics processing and still required low-level hardware communication to perform at an acceptable speed. This need quickly negated its portability and led to the development of a similar concept to the competing FORTRAN P-Stat system in which the Pascal code used a hardware overlay to perform host hardware-based optimisations.

Due to the modern usage of applications, the definition of *portable* has also changed. Compiling applications is now a rarity on any application platform and is relegated to more advanced software users. The majority of users now prefer to use pre-compiled applications, which are easier to install and maintain and are generally distributed after the vendor testing and approval process to ensure maximum compatibility. Additionally, available resources and users' mobility mean that smaller and more powerful computers are now the norm along with the widespread use of multiple machines within assorted working environments. Computer use at home has increased from 36.6% to 58.8% (an increase of 22%) in the ten-year period of 1997 to 2007 with similar upswings in workplace use of 49.8% to 70.8% (an increase of 21%)[7]. This increase in PC use adoption rates has provided consumer markets with cheaper and more portable hardware choices such as netbook computers and mobile phones, which in turn have prompted the increased need for inter-platform portability.

2.2 Existing workspace transference methodologies

Due to the increase of multiple-computer ownership in the general public, it is important to review existing methods of portability in an attempt to understand the current status of workspace transference and to pinpoint its corresponding driving factors and deficiencies.

To ensure a clear review, existing research and technical implementations have been grouped into the following portability categories. These groups have been drawn from the common factors between existing solutions and represent the major portability ideal.

- **Podding:** Transporting the program and data files between hosts in a way that ensures compatibility with each host system. Most commonly employed with smaller software packages that operate without the need for a full admin-level installation on the host machine.
- **Stasis:** Transporting an active program between hosts without terminating the running application.
- **Emulation:** An extension of the podding methodology, this process adds the ability for non–host-compatible software to run via an emulator. An example would be running Linux software on a Windows machine under CygWin or Windows software on Linux using WINE software.
- **Teleportation:** Executing an application on a remote server and providing the visual output of that program on the active host.
- **Virtualisation:** Connecting to or transporting a fully integrated OS enclosed within a virtualised shell. This environment can either be fully enclosed on the current host or run on a remote server via a thin-client connection.
- **System boot:** Booting an entire OS on the host hardware and temporarily replacing the installed host OS.
- **Web-based systems:** Ignoring all local client functionality in favour of entirely web-based technologies.

Following this classification structure, the advantages and disadvantages of each group were examined and the existing research and technology within the area were reviewed.

2.2.1 Application podding

Application podding is the process whereby a program is encapsulated into a portable state, that is, all dynamic references are enclosed within the program, rendering it suitable for moving between systems. This is used mostly with portable USB disks and with applications specifically aimed at mobility such as PortableFirefox[14] or PortaPuTTY[15]. These applications are completely encapsulated, require no host installation, run on most Windows platforms and store data on the portable media disk it runs from.

Within the application podding method are two major industry forces: PortableApps[5] and U3[16]. PortableApps is a collection of encapsulated applications along with a custom installer for deployment on a USB disk or other transportable medium. While simplistic in its implementation, the platform provides technical documentation for programmers to allow their applications to be distributed in this way as well as encouragement from the community at large. U3 is a more structurally viable effort towards application podding. U3 can be viewed on two levels. First, the U3 USB storage media have a slight hardware modification that allows the host computer to detect both the standard USB writable storage partition and a detachable USB compact disc, read-only memory (CD-ROM) drive. This is a minor modification that invokes the Autorun sequence on

Windows PCs from this faux CD drive containing the main U3 boot program. Additionally, the U3 platform enforces a series of standards that a program must follow in order to become correctly U3 certified. Unfortunately, although the CD-ROM Autorun bypass method worked for Windows XP, subsequent Windows releases including Vista and Windows 7 have both disabled this behaviour and now prompt users before executing Autorun applications.

While the U3 development community does provide a set of tools to enable applications to obtain U3 certification, the U3 methodology can be thought of more as a set of principles rather than as an actual protocol. Each U3 program must conform to this series of self-imposed rules set in the U3 deployment specification[17], namely:

- **A program must be portable:** All references, such as dynamic link libraries (DLLs) and other resources, must be supplied with the program.
- **Each program must be responsible for its own configuration:** U3 programs do not exist in isolation. Program A can change Program B's configuration files and vice versa. Therefore, each program within U3 must check and verify its own environmental configuration.
- **A program must not harm the underlying system:** A program *can* make alterations to the system but any changes must be undone when the program is closed or the session is ended.
- **Paths are alterable:** A U3 application programmer should never assume that application file system locations remain the same from one system to another. This obviously implies that the application needs to re-establish its path upon each session start.
- **Packaging is available:** U3, as well as the PortableApps suite, provides a rudimentary packaging system to install U3-approved Windows software from its main website. Both are simply installers that locate the USB disk drive and decompress their payloads into their own respective directory structure.

Program portability is the easiest method to understand from a user perspective. Portable applications such as the above PortableApps and U3 projects provide a simple folder-contained application that can be transported like any other file on a removable volume. The additional freedom of not having a full installation sequence also allows this software to run on computers on which installation is prohibited and/or a user does not possess the appropriate security clearances.

The disadvantage to this method is the concept that the program must run alone and not require any resource that needs to be installed on the host OS. This issue alone can increase the complexity for most application developers, especially those whose products require other libraries that require full installation in order to operate correctly.

2.2.2 Application stasis

Similar to the podding method, the stasis variant executes an encapsulated program within its own individual environment such as a chroot (Linux/BSD), jail (Solaris) or an individual's account (Windows). Unlike the podding methodology, though, when the session ends, the program's state is frozen and saved to disk. When the next session is started, the program is restored from the exact

point at which it was previously saved and the normal execution then continues. This methodology can be thought of as being similar to the Hibernate feature of most OSs in that the random-access memory (RAM) contents, program instructions and other resources are moved between computers without program termination.

WebPod[18] provides a good example of application stasis in action. A browser session (Firefox, in this case) is shown to be frozen in situ, moved between systems and revived on the second host, allowing the user to continue execution without shutting down the original application. Since the Linux architecture (to which *WebPod* is aimed) supports this functionality at the kernel level, there are few problems with transporting a 'live' program between computer systems provided that the architecture is fundamentally the same (with the exception of specialised kernels and hardware). While drastic in its implementation—there are potential problems with transporting programs this way—the overall idea is fully workable. This portability method ensures that a program is completely self-contained and can be moved—along with its memory—with its owner provided that the destination hardware is compatible with the source.

Zap[19] attempts to take this to the next logical step, allowing an entire OS to be put into stasis. The *Zap* prototype achieves this by building on the principles of the older V clustering system[20] and being similar to the *SoulPad*[21] project in that the RAM contents can be 'hibernated' to disk when the session ends.

These three papers outline a workable solution to application stasis and provide a workable solution. However, the requirement of a Linux-based architecture along with compatible kernel and hardware layers makes this portability method highly specialised and limited for the regular majority of non-Linux users.

Some specialised implementations of this portability method, such as the *Internet Suspend/ Resume*[22] system, attempts to implement this functionality using a combination of a virtual machine on top of a minimised hardware layer. While cross-hardware and system compatible, this implementation is limited due to its focus on a web browsers that require no local resources to function. In this single-case implementation of a suspendable web browser, the above paper was successful in its aims, but extending this same technology to other applications would be difficult without the use of a fully featured virtualisation environment (see 2.2.3).

Technological progress with application stasis is exceptionally limited and has largely ceased due to the rise of virtualisation, which has succeeded the practice in favour of a fully encapsulated environment with associated virtual hardware. This environment, as seen below in a review of virtualisation technologies, provides the same functionality as Application stasis without lockdown to a specific OS architecture and hardware as with the earlier papers in this chapter or reduction of software functionality in order to ensure mobility as with the *Internet Suspend/Resume* project.

2.2.3 Virtualisation

Virtualisation is a method by which an entire OS is run inside another physical host system. While slightly cumbersome for the hardware—virtualisation obviously takes a toll on RAM and central processing unit (CPU) cycles, averaging around 20% inefficiency for both—this method provides the most practical portability method in the current application market.

This method contrasts slightly with that of cloud computing to some academics, but its actual definition remains problematic. In the recent paper *A break in the clouds*[23], Vaquero et al. discuss over 20 possible definitions for the term *cloud computing* and draw the conclusion that the definition remains tenuous and unsettled within the information technology industry. However, the paper concludes that virtualisation is only one part of cloud computing and that the two terms are not synonymous. Vaquero extends on this by stating that the concept of virtualisation is, however, a key aspect of what most researchers actually consider to be true cloud computing.

Virtualisation centres around a virtual machine system, such as VMWare[24], VirtualBox[25] or VirtualPC[26], that encapsulates an entire OS and displays it within its own window on the desktop to emulate the monitor output. This method allows hardware separation: since all of the virtualisation OS hardware is 'virtual' itself, a large degree of freedom from local host rights and privileges results. Some virtualised environments (specifically Microsoft's VirtualPC and HyperV systems and Sun's VirtualBox) even allow applications to appear to run in their own environments but to be displayed without the encompassing GUI boundary[27], thereby effectively allowing a program to run as if it were installed on the local host. Under this system, for example, an older program that requires the Windows XP SP1 OS could successfully run under this virtualisation layer while the remainder of the environment runs under the actual OS (or even under further virtual application levels on another OS). This virtualisation system was originally intended for backwards compatibility so that older versions of Microsoft Windows could potentially be extended to non-Windows OSs (OSs).

There are, however, a number of complications with virtualisation. Interestingly, the larger of these impediments is not a technical limitation but an administrative constraint—licensing. As with most new technologies, some companies are more capable than others in this area. The central precept is that since there is no physical lockdown to a given host, software becomes much more easily cloned. This leads to the possibility of two copies of the same virtualised OS being run by two different people after purchase of only one license of a given application. This is an uncomfortable situation for some publishers and has led to some companies to dismiss virtualisation altogether and simply refuse to distribute licenses for their software for these kinds of systems. Such restrictions create large obstacles for the adoption of virtualisation and can represent some of the more debilitating industry factors in this area. Further complications occur when large numbers of cloned virtualised environments are legitimately required. Universities, for example, have a need to educate a number of students working within the same environment. The complexities of achieving this set-up can be largely eliminated using virtualisation. For example, a pre-configured Windows environment with a complex Oracle database (DB) engine, Microsoft Office and other software packages can

be configured once by the system administrator and cloned for the number of students requiring this basic set-up. Traditional licensing models are based around the simple 'one piece of software requires one license' system, which raises the obvious question of whether the above scenario requires one license duplicated many times or many separately registered licenses. The solution to this complication is a corporate key—a single key that allows a set number of distributions, thus making the cloning process much easier.

This problem is multiplied by the digital rights management systems that these software packages usually employ. Microsoft Office, for example, does not allow a license change without a complete reinstallation. Thus, a set-up not using a corporate mass-distribution license would quickly encounter problems when deployed more than once.

In *Controlling Data in the Cloud*[28], Chow et al. discuss the above policy, legal stipulations and three major areas preventing widespread virtualisation adoption within larger industries:

1. Traditional security
2. Availability
3. Third-party data control

While not indicative of individual users this list still sets a suitable benchmark for the assessment of virtualisation technologies. Security, for example, is of less concern to the average user but is a prime factor for larger corporate or group users. Availability is the concept of connecting to these virtual environments (see 2.2.4) in which the concept of the user being physically separate from the virtualised machine becomes a factor. Finally, third-party control discusses the licensing and control factors previously discussed along with the additional issue of virtual machine (VM) compatibility. For individual users, points 1 and 2 above are largely irrelevant or accepted during the physical transportation of the environment within portable media, while point 3 becomes a factor only with regard to the specific software running within these environments.

Virtualised systems in themselves can be highly portable and personalised OSs that allow full transportation across hardware and host OS boundaries with relative ease. When coupled with other technologies such as remote access via a remote server (see 2.2.4), these solutions can take on further advantages at the expense of some deficiencies (e.g. constant connectivity).

Separate from the concept that a virtualised system should be completely self-contained are research projects that allow for separation of the virtualised OS from individual applications running within these segmented environments. One such implementation is VMWare ACE[29], which encapsulates a single program within a virtualised operating environment and provides rudimentary application portability across system boundaries. While aimed only at the Windows OS, this environment allows single installation of the software, which can then be transported as a proprietary image onto another Windows machine and run immediately without the need for subsequent installation.

Another interesting variation on the concept of single program encapsulation is discussed in *The Case for VM-Based Cloudlets in Mobile Computing*[30], which proposes a portable cloud-based

architecture that can travel with the connecting agent via nearby hardware (in this case, *cloud* is defined as a single application). An example of this concept would be a lesser device such as a mobile phone requiring an advanced feature not normally available with its resources, such as voice recognition or transcription. Using this process, the cloudlet virtual machine (as referred to in the paper) can be used within a local VM and connected via a wireless local area network. The speech data could then be transmitted from the phone to the intensive cloudlet and the transcription could be sent back to the phone. This whole process extends the normal practice of outsourcing the processor and resource-heavy actions not normally available on the client hardware with the added advantage of the cloud computing resource being semi-mobile. Previous cloud computing methods involved a fixed location—usually a VM farm data centre—but this method also reduces bandwidth costs, often the largest obstacle for bringing virtualisation into a remote data centre, by enabling the virtualised computer to be transferred onto a local resource such as a company-owned VM server.

Along with the increasing focus on environmental policies, VM technologies can be used to migrate large, once-physical data centres around the world to more suitable environmental locations. This concept is detailed in the paper *GreenCloud: a new architecture for green data center*[31], which proposes such measures to leverage environmental factors such as natural weather cooling. These technologies are, however, outside the realm of a regular user and are primarily focused on larger businesses that can afford both the extensive bandwidth and virtual provisioning involved in the process[32].

The format and security integrated in these virtualised environments are proprietary for the most part and are, therefore, limited to the vendor's implementation. While some third-party solutions have been discussed—such as the authenticate/response method proposed in the weblet (small web-styled components) model within *Securing elastic applications*[33]—these implementations are usually accompanied, as in this case, by full implementation of the VM architecture and are rarely usable within vendor-supplied solutions.

An often-claimed advantage of virtualised platforms is their inter-VM migration. Most large VM farms (a collection of larger servers running multiple machines and sharing data) provide for this functionality via optimised migration of the image and RAM contents. This method provides load balancing between each of the hosts in the VM farm as well as single-node failover support, but the optimised networking and shared disk access (usually a storage area network or network-attached storage system) in hardware is often reserved for higher-end data centres. Likewise, the bandwidth between these machines that account for their constant synchronisation to perform immediate VM transfers can be demanding on the standard networking infrastructure[34].

Virtualisation under a single OS can provide a number of advantages to the user, including hardware and OS portability, but the strains on the local host can be large and the resource allocation required to actually store the VM can be larger than normal storage media allows.

As for other virtualisation downsides, the encapsulation of a full virtual OS within its own large window/full screen control area can be a problem when software is required on the host machine. The slowdown presented with the virtualisation environment inefficiencies can be a problem

when efficiency is required, but as discussed above, the most pressing problem is the software manufacturers themselves, who are slow to adopt this method of license distribution.

2.2.4 Teleportation

Teleportation is the process of retaining the processing on one centralised machine but allowing movement of the application's visual output to a variety of potentially mobile displays. Like virtualisation, this relies on a single larger server that is for processing the system background, whereas a smaller client machine connects and displays the graphical outputs.

Being the most obvious and established platform, the majority of these papers are based on the X windowing system[35], the *de facto* standard GUI of Linux-based systems. X implements this practice using the client-server method, in which a server performs the processing and the connected client displays the visual components. Since the actual protocol can be carried across any communications transport, a strong cipher such as the Secure Shell (SSH) protocol is generally employed to provide a secure connection to the host.

Based on the X technology, *Teleporting*[36] and its younger sibling *XMove*[37] provide compatibility for other OSs with the requisite client software. An example of this is the CygWin[38] emulation layer that provides X-client functionality to Microsoft Windows, allowing the client system to display X-compatible visual GUI elements that can be rendered according to local display characteristics such as font size or style.

Easy access to remote graphical UNIX applications for Windows users[39] discusses the process of implementing CygWin within the Windows environment, which can be arduous to maintain. The paper instead recommends the use of a pre-prepared set-up such as the XLiveCD[40] project provided by Indiana University, which supplies a pre-packaged CD-ROM image that allows CygWin installations for less able users. The XLiveCD project drastically removes the complication of establishing a standard CygWin install, X-client functionality and Portable OS Interface (POSIX)-compliant console access regardless of the host Windows OS factors.

Older protocols such as the SLIM[41] and VNC[42] systems provide a full streaming image of the host machine to the client, effectively replicating the behaviour of connecting the monitor output of these machines to the client. While workable, use of this protocol means that any visual element is actually rendered by the host machine (using its styles drawing methods)[43] and then merely transferred to the client. Using a system such as X or the Microsoft Remote Desktop Protocol[1] allows for selective elements of the remote environment to be displayed (e.g. just one program rather than the whole screen) and any visual element is actually rendered by the client and not the server. For instance, to draw a standard button, X/RDP can simply send the directive of where to place the button on the screen and the properties affecting it such as the caption or font size. Actual rendering takes place on the active client machine and follows the styling, accessibility and optimisations present there rather than on the server, which now becomes responsible only for the processing component of the program. The selective display and rendering process ensures that X and RDP

both result in larger bandwidth savings[44] and seamlessly integrate a remote environment into a local one.

This system allows for a larger server-based computer to concentrate on processing and, when used with a system such as virtualisation, can improve performance in processing - assuming the client is responsible for rendering only and the dedicated remote server can be optimised for the core processing tasks.

Unfortunately, this system has a single bottle neck as a weakness, namely the connectivity that must be maintained between the client and the server. While allowing clients to change as often as needed is practical from the user mobility perspective, each client machine must have a usable network connection that can be used to transport the required interface information between the client and the server. This is not always possible and can represent a large deficiency in areas of limited connectivity.

2.2.5 System emulation

This form of portable computing centres on the owner transporting a series of applications in the manner of application podding (see 2.2.1) but providing a translation layer for non–host-compatible applications.

There have been some attempts to capture this method of system portability that have been executed with varying degrees of success. *MojoPac*[6], a pseudo-virtualisation and -emulation system, implements what is essentially a root-kit for Windows. MojoPac installs its own child process over the top of the active Windows environment, creating a fully encapsulated OS with all the advantages of a boot disk system except for the hard disk access; therefore, all of the configurations set up on the parent network drives are still accessible. This approach allows for local resource access (e.g. three-dimensional acceleration) but still provides a portable environment with which to encapsulate a user-customised OS.

Some other attempts at alternative OS emulation include *MiniVMac*[45] a Mac OS 8 emulator, and *WINE for Windows*[46].

The WINE project (in the typical ‘GNU is Not Unix’ [GNU] acronym style: ‘WINE Is Not an Emulator’, although some prefer the more helpful but contradictory ‘WINdows Emulator’) initially started as a method to run Microsoft Windows software on a non-Windows platform. Since its initial start-up days in 1993, WINE has developed considerably into an almost exact duplication that sometimes even rivals natively installed Microsoft Windows instances by producing higher performance benchmarks[47]. Since WINE can effectively run Windows applications in most environments, it is logical that such applications can be executed within Windows itself. An application can potentially be emulated via WINE with its state, registry settings and other information being saved within the standard WINE storage format on any supported platform, allowing for portability of Windows applications within this emulated pseudo-Windows environment.

While WINE does perform favourably in some systems, it does suffer in others, especially with regard to the obvious performance reasons of running an emulator within the system it is supposed to emulate. Additional problems with this approach include side effects such as program crashes (for less supported or tested applications) and the unfortunate functionality of providing a transportation vector for Windows virii into other OSs[48].

A separate console-only approach is shown in *Personal Information Everywhere (PIE)*[49], which attempts to make a low-resource terminal emulator for server interaction. PIE is primarily aimed at the personal digital assistant platform and attempts to manage with the small resources on hand and emulate a thin-client environment. Similar to this are many emulation environments including the *Feather-Weight Virtual Machine*[50] and the more popular *XEN*[51], both of which are based on the Linux environment. Linux virtualisation is not a new discipline and is a keenly contested area of OS development with regard to emulation, but few concessions are made for other OSs and the overheads employed in these systems can be large[52]. Projects like FVM, XEN, *coLinux*[53] and *User-Mode Linux*[54] provide both the ability to run multiple OSs within encapsulated environments and the ability to teach students about the inner workings of such systems without giving away host system Administrator privileges.

Under these systems, an operator can have his or her own personal computing environment to experiment with complete with full Administrator privileges to that personal environment. Should such a system become unusable, an image can simply be restored from a backup, which typically takes the form of a binary device (representing the inner system hard drive image) being overwritten with the last known working image. In addition to the educational aspects, these systems also provide the ideal environment to observe disaster situations such as virus activity and major system failures[55] without endangering the host OS.

Emulation provides many distinct advantages for portability, specifically for applications compiled for other OSs, but its use in portability should not be underestimated. Emulated environments are often self-contained and can provide the ability for any contained application to be transported in a safe and portable way without needing a full OS to enclose it.

Unfortunately, emulation suffers from sometimes unpredictable results. Larger companies such as Microsoft are generally loath to reveal their proprietary library formats, and WINE development in particular has largely involved guesswork and system call debugging. The result is that the emulation platform is sometimes unstable, namely where a given result under the target system may not exhibit the same behaviour on the emulated one. As such, while desirable, this approach may not work in all cases.

2.2.6 System boot

A system is loaded with a bootable CD-ROM that the computer boots into rather than its natively installed OS. This is the most commonly used method for privacy protection since no data are actually saved on the local system.

Due to the architecture of the Linux kernel, this method is already supported by the top three Linux distributions[56]—Ubuntu[57], Fedora[58] and SUSE[59]—providing a variety of methods to sample the OS before full installation. Specialist disks such as those geared towards forensic recovery are also popular in this area to repair or restore data on an otherwise non-functional machine.

This method is not limited to CD-ROM boot methods but is also open to any supported media such as writable external hard disks (serial advanced technology attachment or USB). Using such writable media gives the added feature of being able to save back to the system image, allowing applications to be installed and retained after shutdown[60].

While highly portable, there are a number of issues with this environment. This method currently functions only on Linux-based systems (although a rudimentary FreeBSD project is in the early testing stage[61]). Linux can be an unfamiliar OS to some and the learning curve for low-ability Windows users can be steep. The booting process is greatly extended since the OS has to detect hardware and locate matching drivers each time the system is initialised. Additionally, not all hardware is supported, which can cause problems for more customised systems.

The system boot method is also difficult to set up and maintain. The process of installing a Linux distribution to a USB key, for example, is quite arduous and is really only recommended for advanced users. As such, it remains outside the ability level of the typical user.

2.2.7 Web-based systems

The emergence of Web 2.0 technologies has led to not only a huge amount of technological innovation with online content but also the reanalysis of standard business models. Web innovation has been driven largely by the low entrance costs for hobbyist or small businesses and have lowered the barrier for service-based businesses[62].

Typically, a web-based system would comprise of a solution to a single problem area whilst containing a psudo-standard of typical actions such as user login/logout functionality, email contact and usernames.

More complex systems also leverage application program interfaces (APIs), which allow interaction with various content aggregators, desktop clients or mobile platforms for further client-end interaction. This free exchange of information has lent itself to extremely vertical business models, a product that essentially performs *a single function* but with all the specialisation that a user can expect from a dedicated piece of software.

These solutions are being adopted at an increasing speed, sometimes to the degree that existing established business practices are simply inadequate to serve these newer models. Most modern Web 2.0 solutions provide a two-tiered service, a simplified free service and a secondary premium service with more functionality. The intention here is to coax the user to migrate from the former service to the latter paid service.

It should be noted that web-based systems present both a huge advantage and an equal disadvantage. Web-based systems correspond with all the early portability methods, even going so far as to render some moot, namely hardware and data portability, neither of which affects web-based applications. Unfortunately, though, web-based systems are limited to online access only and there is an underlying assumption of an active Internet connection. There are various ways around the issue of offline access such as Google Gears[63] or the new HTML5[64] standard, both of which propose demonstrated methods under which data can be stored locally and retrieved by a cached mirror on the offline host without requiring an active and constant Internet connection. However, both of these solutions are in their beta phases and are limited to a short-term workaround; therefore, neither can be considered anything other than a stop-gap to the problem of inherent client–server-based Internet communication.

While it would be desirable for web-based systems to operate with offline functionality, the current technology is not able to totally replace a local application environment.

2.2.8 Summary of workspace transference methodologies

From the above review of the existing academic portability research, it is clear that while some individual components of true portability exist in a rudimentary form, a unified and concise encapsulation of the concept is still missing from the relevant literature.

Two of the above technological implementations in particular stand out as being close to workspace transference: virtualisation and web-based systems. Virtualisation is a rapidly evolving technology, primarily due to its accelerating prominence in computing and server consolidation schemes. Because of this rapid growth, the technology is either becoming increasingly capable or is virtualising more and more complex systems within these virtual hardware sets. Most manufacturers now provide client workstations on each OS, meaning that the individual VM images can be easily transported across the OS boundary along with the internal virtual hardware requisite. For portability, virtualisation can be applied in two ways: on a server, which emulates the issues of the above teleportation method; or on a client machine, which emulates a cross between application podding and emulation. The former method presents all the same issues that are present with the teleportation method, namely that a constant connection to that server is required for the client display (RDP, VNC or some other communication protocol) to function. With the latter method, the same problems are present as those of the application podding method, namely a single location where the data is centralised (i.e. the same machine as the processing entity), leading to data loss if that single location is ever compromised.

These issues underscore the lack of a workspace transference solution in the current research and provide a case for analysis into this area.

2.3 Other portability issues

In addition to the above portability methods overview, a number of similar issue areas are present in many of the above portability methodologies.

This section discusses the common issues of individual file synchronisation, DB synchronisation and security concerns and reviews the existing research and commercial implementations available for each case.

2.3.1 File synchronisation

Separate from the major portability method in use is the concept of data sharing between individual machines representing the workspaces of an individual user. While the actual workspace may be a home computer, laptop, workplace or office computer, the individual data files representing the user's work should be equally portable.

The typical method used here is to transport the data files with the user, merging the concept of data portability with that of the traditional physical transportation methods inherent in a person simply transporting a piece of paper containing the same data. While practical in the short term, this concept is vulnerable to the traditional means of data loss, namely the data storage media becoming damaged, lost or stolen[65].

The alternative way of transporting data between machines is to abandon the traditional portability of physically carrying these data storage devices and to move to a completely digital method whereby data is moved between various workspaces.

Synchronisation of files across multiple hosts is a relatively mature technological concept but one limited to some arbitrary uses[66]. Primarily aimed at the more able computer user, most synchronisation takes the form of DB[67] or file transfers using peer-to-peer protocols between individual nodes. Some more advanced DB systems have dedicated master-slave relationships to optimise the load organisation.

While there are a number of commercial products available in this area, the most comprehensive and *de facto* standards include the RSync algorithm[68] and the older Diff[69] functionality. Being a central backbone of the open-source software movement, it is hard to find a project involving large-scale binary transference without eventually encountering the RSync algorithm or a variant thereof[70]. The older Diff-based systems tend to form the backbone of most code management and plain-text systems that require merging of multiple user contributions, whereas RSync can manage text and binary data.

More often than not, file or workspace synchronisation has been lost for lower ability users, but some recent attempts have successfully made such a complex procedure more accessible. Unison[71] and Microsoft's ActiveSync[72] are notable works within this area. These two projects are the larger and most widely populated solutions available today. These two solutions come from radically different methodologies and follow two different paths to the same ultimate goal.

The process of file synchronisation is an interesting one that could form the backbone of any proposed portability solution. Peer-to-peer synchronisation also offers an optimised method of carrying the data differences performed by the user rather than the whole data file each time a change is made and supplies an optimal method of tracking data across multiple workspaces.

2.3.2 Security

While not the central focus of this research, the concept of security policy portability has gained interest due to the adoption of virtualised or cloud-based data centres that can use a variety of mixed OS platforms.

Some file systems do support security policy replication across OS boundaries, but this support is largely dependent on two major issues: 1) support of the underlying file system to contain such security information and 2) the host OS's obedience to these policies. An example of this would be the new technology file system (NTFS)[77], which has the ability to store and replicate complex access control lists, which can in turn control access to files and applications. Unfortunately, since the NTFS is proprietary in nature, replication across other platforms such as Linux or Macintosh is largely *ad hoc* and leads to the host OS ignoring the security policies specified by the file system.

Some attempts have been made to promote the portability of these security permissions; most notably the recent Policies & OntoLogies for Interoperability, Portability, and autOnomy (POLIPO) proposal[78] attempts to use a portable ontology system to support these security protocols across operating and file system boundaries with little implementation needed at the file or disk level.

Security is not a central focus of this research, but the prospect of existing security solution portability is an interesting one and may form the basis of future work in this area.

2.4 Literature Summary

The literature examined in the examined papers does somewhat address portability issues. However, these areas are minor and are available only to the more technically able user base. The virtualisation method (see 2.2.3) in particular requires reasonably advanced knowledge of how to set up one OS inside another and the expertise to set up such a system in a reasonably portable way. PortableApps[5], in contrast, provides a relatively easy method for the less able. However, both methods suffer from the usual deficiency of data loss should the storage media be damaged.

Of the above methods, the web-based systems approach currently has the most support. The recent re-emergence of the Internet bubble has provided a free market system in which the entry costs and overhead are low but the potential worldwide profits are high[62]. This business-led revolution within the industry has formed many diverse companies that could previously not compete against the Microsoft and Sun mega-corporations of the world. It is now possible for a single developer to create a small web-based application and have the Internet provide the infrastructure to allow users to access the service via a simple and ubiquitous distribution channel. While desirable in that any potential web-connected terminal can become a workplace (after successful login to the relevant service), the data are not held by the user and the proprietary lock-in by these different vendors can be undesirable as the work created in these systems increases. Additionally, local services such as locally installed software or hardware or file stores remain more or less unavailable to these pure web-based systems. Some solutions are being developed to address these problems (HTML5[64], for example), but their widespread adoption is not yet near.

It can be concluded that each of the above solutions has implementation gaps. Some provide various elements of workspace transference to a set degree, but most suffer from either a steep learning curve or a single point-of-failure. The method that has gone the furthest to combat these problems, web-based systems, solves the portability issue since it does not require physical media but lacks the ability to interact with local hardware in the same way that the Emulation, System Booting and Application Podding methods provide. Each of these research papers can be seen as being logically disconnected and isolated from one another with no unifying method of combining the existing research to create one coherent outcome and solve all of the gaps that other papers may have already closed.

A potential solution to these deficiencies can be drawn from studying the above papers and technologies with a view to understanding their core competencies and constructing a unified list of these aims. The matrix shown in Table 1 attempts to illustrate this concept by comparing a number of existing solutions against a list of core aims that are discussed further in the next chapter (see Chapter 3).

		Application Podding			Emulation		Teleportation	Boot OSs	Virtualised Systems	Web-Based Systems
Specific Technology	**Section**	**PortableApps & U3**	**WebPod**	**Zap, V & SoulPad**	**MojoPac**	**WINE**	**RDP, SLIM & VNC**			
Application portability	3.1.1	✓✗	✓✗	✓	✓	✓	✓✗	✓	✓	✗
Data portability	3.1.2	✗	✗	✓	✗	✗	✓✗	✓	✓	✓
Data sharing	3.1.3	✗	✗	✗	✗	✗	✗	✗	✓✗	✓
Environmental interaction	3.1.4	✓✗	✗	✗	✓✗	✓	✗	✗	✗	✓
Interface requirements	3.1.5	✓	✓	✓	✓	✗	✓	✓	✓	✓

Table 1. Matrix of existing research compared to the workspace transference framework requirements.

Through examination of Table 1, it can be concluded that the main concentration of existing work has supported data portability between environments at the expense of some of the other requirements. In particular (except for two concessions), data security is often overlooked and is an area of portability that obviously needs more attention. Likewise, environmental interaction and settings are covered in only a few selected papers and have obvious research gaps.

From Table 1, the following conclusions can be drawn:

- Application portability is largely an established research topic and requires little extension from a research perspective.
- Data portability is a moderately settled area with the exception of two specific technologies (PortableApps and WebPod under Application Podding).
- Data sharing is largely ignored as a requirement in all but the web-based system.

- Environmental interaction is covered only in some areas and requires closer examination within the literature.
- Except for two notable exceptions and methodology (Teleportation and Web-based systems), most existing methodologies succeed in being largely offline compatible as per the interface requirements.
- Most existing research allows for an acceptable ease of use degree.

This outline of each of these aims can then be applied towards constructing a framework. After understanding these aims and defining what each achieves, it is possible to understand how each existing research paper can assist in the overall goal of workspace transference using this existing research in combination with new approaches in order to provide a stable solution.

The next chapter will fill the research gaps discussed in this literature review with the creation of a workspace transference framework.

Chapter 3. Workspace Transference Framework

The analysis of research within the literature review in Chapter 4 determined that some potential literature for workspace transference is available, but no solution currently exists that contains all of the workspace transference requirements. To counter this deficiency, this chapter describes the design and implementation of a workspace transference framework. This framework draws on the existing overlapping solutions listed in the literature review and attempts to provide an amalgamated solution that unifies these solutions under the workspace transference definition contained in the research question (see 1.2).

The overall aim of this framework is to forward the ideals of workspace transference in order to allow application and data portability between physical host machines (in this research, *host* refers to the physical computing device on which the user is working). The intention is that a user should be able to operate any number of physical computers yet still maintain a consistent working environment regardless of any active host machine limitations.

In accordance with the framework specification discussed in the research question (see Chapter 1), the below aims illustrate the various framework goals. These goals constitute a list of requirements that must be satisfied by any research outcome attempting to demonstrate the concept of workspace transference.

3.1 Framework aims

The workspace transference framework has been drawn from the existing literature and technical solutions listed in the literature review. The framework is intended as a gathering of the common features of each of these existing technical works and academic literature in order to form a solid set of aims and requirements for workspace transference.

The list below shows the workspace transference requirements listed as the components of the workspace transference framework. As per the analysis in Table 1, the goals enumerated within this framework address both the research outcome and any existing technical solution.

Workspace transference framework requirements

- **Application portability:** An application can be moved between software and hardware environments without any significant interaction with the user.
- **Data portability:** Data can be moved between environments without any loss or visible conversion, again with no involvement from the user. This requirement also ensures that data is secure while being transported between these environments and that data is restorable from any point within its lifetime (employing a versioning system).
- **Data sharing:** If desired, the managed data can be transparently and concurrently shared with other users. If two separate users are editing the same document, the changes should be seen concurrently by both users.
- **Environmental interaction:** System-specific configuration can be accessed from within the portable environment in a consistent and cross-platform manner. Additionally, any

access to hardware, printers, network shares or other applications running within the host environment needs to be available to the portable workspace as well. This is broadly the transparent pass-through of information to the underlying host OS without requiring a complete software reload. An example would be a word processing software having access to a list of printers regardless of the physical machine the software is running on. This should be achieved *without* requiring any action on the part of the user to re-detect or re-configure such hardware.

- **Interface requirements:** The solution should be capable of operating without a constant connection to the Internet and present a suitable user interface (UI) during operation.

The above list briefly enumerates the framework requirements. Each of these points can now be examined in greater depth including an analysis of the history of existing technological solutions (if any), available academic literature, portability ideals and the justified need for that portability requirement.

3.1.1 Application portability

As shown in the literature review, the research surrounding application portability is slowly expanding due to recent developments in virtualisation technologies. However, existing research has not concentrated on individual programs being moved between machines but rather between OSs. Differential synchronisation such as VMWare VMotion or XEN Live Migration[79] and other high-availability/failover technologies effectively move the entire working environment between larger virtualisation hosts. The actual object being transported in these cases between systems is the OS itself comprising the disk drive(s), RAM contents and other hardware memory stores (IAM or register contents within the CPU hardware layer, for example). Although the relevant portability technology is maturing quickly due to its adoption within larger companies as well as the recent upswing of cloud computing technologies[1], individual application transference remains limited.

The application portability requirement enables effective movement of applications between machines with no need to exit and re-establish the program's state (start-up and shutdown). Regrettably, while the full extent of this requirement would be preferred, existing limitations at the OS level make support of this functionality quite limited for complexity reasons. Due to these existing technology limitations the transfer of executing programs falls outside of the remit of this research.

An additional issue is one of environmental or hardware concerns. An application written and compiled within one environment does not necessarily transfer equally well into another due to hardware and OS differences. This is a well-known and established issue with most OSs and hardware platforms but is of little interest to the user base.

The inclusion of application portability as a framework requirement should ensure that an application should be transferable across these boundaries. A boundary in this sense being defined as either another OS or some form of physical hardware separation. A program should function equally well on one machine as it does on a second without requiring interaction from the user.

The need for application portability is one of user convenience. A distinction is made in computing terminology between system-specific programs and programs performing a general function. The former group is referred to as utilities, the latter as software. While any computer program under the utility heading can be OS- and hardware-specific, the software category should aim not to be. This explicit definition should attempt to provide maximum portability for anything that is classified as software, these programs being the prime motivation of using a computer by a regular user. Software that functions as a word processor on one machine should function equally well on another. A word processor's central goal, after all, is to allow a user to type a document. There is nothing in this purpose that restricts this software to one platform, environment, OS or hardware specification unless it is intentionally designed to do so.

To achieve these aims, some entity must be created that provides this portability layer to the contained applications. This entity should attempt to provide the same portability as the contained applications, although such functionality may be limited under certain circumstances. Within the application portability framework requirement, any conforming application should be portable between these environments and allow an operator to use it. A 'conforming application' in this context is any application that runs within the entities portable environment.

While some major application vendors are now moving to a unified distribution system supporting multiple platforms, most still release platform-specific applications. The reasons behind such a single-platform distribution method are historical and relate to the difficulty of releasing an application that could cross these boundaries without any specific OS or hardware requirements. In the past, with the availability of less capable cross-platform programming languages, software propagation was limited to larger group or commercial efforts and typically employed an 'engine' that would provide the actual functionality part of the application layer within a common environment. An example of such an engine in action would be a word processor that provides cross-platform compatibility to maximise its sale potential. Time must be invested to develop a mechanism to draw and manage the program’s GUI, the graphical resources for each OS or hardware environment and the specific functionality required to store and retrieve files from that platform. Most if not all of these problems have been handled by either cross-platform or cross-hardware toolkits (in the case of GUI functionality by such efforts as Wx[80], Tk[81] and Qt[82]) or adaptations to the application programming language that provide a universal front end to most file and graphical operations. Likewise, most OSs have attempted to provide a clean and functional set of APIs that remove some of the more tedious areas of programming from the development process and place this functionality under either the OS or an intermediate layer capable of performing the environment-specific actions.

The recent upswing in scripting language popularity has provided a number of advantages for cross-platform compatibility. It is no longer necessary for an application developer to code an engine and the actual central application functionality. The engine component, complete with GUI, graphical, file handling, garbage collection and other system- and hardware-specific functions, now largely removes the need for anything other than the development of the application core function. The aim here is that any script created within these environments on one platform should be capable

of running the same script on another platform with little difficulty unless the script designer is specifically calling on some hardware- or OS-level functionality.

Unlike software portability, hardware portability contains some unique pitfalls that could limit application use on separate platforms. More drastic hardware changes such as limited screen size, RAM or CPU limitations and lack of an adequate graphics processor for graphics-intensive software can all limit software portability from an adequately resourced platform to a less able destination platform. As a whole, however, more demanding applications tend to be the exception rather than the rule.

Application portability should be supported within the framework in order to provide transparent application transference between computing environments regardless of the software or hardware makeup of any host hardware involved in the transference process.

3.1.2 Data portability

While data portability was once thought to be the domain of larger companies with access to technologies such as Microsoft Active Directory[83] or other distributed computing resources, the concept is quickly becoming the norm expected by cloud computing adopters.

A file can represent any amount of binary or text data that is transferred between machines or to a central location (with an attached backup). After this transfer has occurred, any number of subsequent steps can be conducted to further distribute the data amongst peers. The ease of copying digital file contents represents a significant advance in computer technology as opposed to older replication systems including vinyl records or carbon paper. Digital data can be replicated any number of times without loss or deterioration, allowing for one single file to become any potential number of duplicate copies.

The idea of moving digital data around with the user has largely been bound to the older thinking of physical transportation. Like carrying a physical piece of paper, users carry data on a USB disk or other portable digital storage device. While practical in the short term, any upset to this medium carries with it the same issues of the piece of paper: the data device can be damaged, destroyed or misplaced. Even non-mobile storage devices should be regarded with the same caution. While a single desktop computer fixed to a physical location does not move about and expose itself to some of these dangers, it is not immune from the issues of data loss via physical hardware failure, viral infection or other such destructive events.

The inclusion of data portability within the workspace transference framework is important since it allows a user access to the file set regardless of physical location. Data, even an entire workspace, should be available wherever a user is located rather than requiring a user to carry physical storage medium.

While some computers without an Internet connection remain, they are quickly becoming rare. The natural assumption here is that the physical devices used to carry data will become equally as rare.

Unfortunately, most users are dependent on this data storage method, usually because adoption of a truly portable and non-physical data storage method requires technical proficiency outside a regular user's abilities. The functionality expressed in this requirement is a different paradigm of data transference between machines and, hence, is immune from the dangers of single-location storage.

An additional concern if data portability is realised is that the data being transferred must be secured during transport between host machines. A degree of security needs to be demonstrated within data portability transfer protocols. Ideally, a standard that is open, tested and respected for its end-to-end security ability will be incorporated. The end-to-end security requirement itself attempts to strengthen any solution following the framework from trusting a potentially compromised middle-tier communications node that may be intercepting the data packets between the source and the intended destination. This is especially relevant if a user is working from a less trusted computing environment such as a public terminal or over an untrusted or unsecured Internet connection.

An additional feature should be the ability to restrict the data from host to host. For example data synchronisation should be allowed from the office machine to the home machine (if the user wishes to work from home) but the reverse can be prevented (if the user doesn't want to burden his office machine with personal pictures for example).

Data versioning should also be required under the umbrella of data security. Since this functionality usually requires a large amount of storage space, it should be conducted on a larger server that has the ability to automate and optimise the process. This versioning functionality allows for file retrieval from any point in the data life cycle up to and including deletion. It is provided as a means of both a fall-back solution should the user accidentally lose or overwrite data and a second level of defence against data loss that has the potential to occur at any point in time.

The need for such security is also relatively obvious. If the user wishes to trust his personal and private data to any solution within the workspace transference framework, it must be established early in the design process that the managed data are secure and that data transfers occur only to workspaces authorised by that user.

3.1.3 Data sharing

To frame the data sharing discussion, it is first necessary to define its boundaries. The manual moving of a series of digital files between users (e.g. emailing a file back and forth) does allow users to work on the same data-driven project but does not meet this requirement's definition of data sharing. While practical in the short term, there remains the consideration that group members may fall out of sync with the original source material. For example, the traditional method of working on a programming project amongst a group of developers usually requires all members to select a problem area and focus on that functionality. This typically takes the form of the individual modules being distributed amongst a number of programmers. Issues arise when interaction between these distinct modules is required. More established methods of design such as Structured Systems Analysis and Design Method, user-mode Linux or top-down design principles enabled the project

scope to be defined before development began[84]. This then allowed for effective documentation of the required APIs and other such bindings between these diverse modules.

Modern design principles such as Agile programming or rapid application development can, depending on the project philosophy, abstain from this up front API standard and instead use an *ad hoc* method of coding to each of the project team's strengths. This, unfortunately, requires more or less constant updating of each group member's code base to align it with that of the others.

In the above example, data sharing exists simply to enable a large group of users, in this case, a more able developer user subset, to work on one large project in which each member can potentially have a concentrated or random access approach to coding across the entire project. Various version control systems have evolved over time to solve this deficiency but most rely on the older pull and push methods of updating (i.e. a developer would invoke the update from or to the server, respectively). This method enables the project to continue, but the failure of most of these established code management systems to support functionality such as branching and joining can quickly prove burdensome for coding in large groups where each module's API can change almost daily and each member must have the latest version of the other members' work.

The above example is a complex but established problem and relates specifically to a user group with a higher technical ability level than most, but the same working principles are often present when a single data source can also be the project bottle neck. Examples include data being passed via email, updated and sent back or data stored on a networked and sharable volume, but these modalities feature the same limitations as the legacy data sharing practice of a single piece of paper being passed around an office.

Within the context of this research 'sharing' is defined as a single user providing access to resources owned by the user. Thus a file can be edited by multiple individuals but is still the propertly and domain of the original sharer.

This framework requirement must allow a large group of users to access, update and immediately see all users' changes without use of a manual data refresh. It would also ensure that any data contained within the environment would allow immediate access to and from other group work, regardless of whether the shared data is located in a different physical host, office or network.

3.1.4 Environmental interaction

While the ideal workspace transference system provides a contained environment separate from the host OS, local configuration settings support is needed for this environment to function. One such example is the host system networking settings that specify the various required configuration for any software to perform various networking tasks, such as the proxy server settings that are used by hypertext transfer protocol (HTTP)-based requests to establish connections with remote servers. Without this information, HTTP connections would simply fail, rendering most software depending on them unusable. In order to work around this issue, most OSs support a system-wide repository of such information that is accessed by the individual software requiring these connections (see

3.2.3 for further discussion of system-level settings). Cases such as this underline that, while a self-contained portable workspace can be constructed, it is inevitable that some information will be required of and should be provided by the host OS.

Any software requiring specific information from the host OS should have a unified method of retrieval. Each OS stores this information in a specific way: Windows, for example, stores its system information in the Windows registry, whereas Linux-based systems store it in either the system variables or the configuration files. The retrieval method of the workspace transference framework must work with these different and diverse methods of storing information and provide it to the contained workspace software.

The inclusion of environmental settings within the workspace transference framework provides a cross-platform portal allowing the host OS to provide needed information to the workspace software regardless of the actual storage and retrieval methods employed to access the information. A self-contained environment provided by the framework is a workable solution within the area of workspace transference, but unless such an environment can access local resources, the contained applications will be limited to their own ecosystems of available hardware, software and other provisions shared by the host system.

Some advanced computing environments can sometimes access hardware connected specifically for that local system's dedicated use, for example, sonogram and X-ray machines used by medical professionals. Neither of these hardware devices is available to all machines; instead, each is bound to a single dedicated computing environment. Likewise, specially licensed software running on one dedicated machine can suffer from limited mobility. In these cases, the transferred workspace should be able to take advantage of local software and hardware without negating their abilities.

Software available to the host machine should be available within the user's personalised environment and easily interface with existing software. The need for this framework stipulation is mainly limited to environments where local resources are needed by users, such as interactions with specialised hardware or software (as above), local files, printers, devices and setups. This framework requirement ensures that, while portability is supported, such environments are not entirely segregated from the outside world and interaction with local hardware and software remains possible.

3.1.5 Interface requirements

While not the central tenet of the workspace transference framework, it is necessary to stipulate that any implementation created in adherence to the framework should be easy to use to ensure effective adoption. Acknowledging that this ease-of-use framework requirement is quite subjective, it is proposed that the two extremes of the user ability spectrum be tested against the implementation outcome as well as an intermediate middle level. Ease of use is needed to attempt propagation of the framework amongst low, medium and high ability level groups and must be provided for if a solution is proposed to conform to the framework. To satisfy this requirement, software must feature ease of use with both the lowest and highest ability levels.

As a result, any solution constructed following the framework requirements must be tested against the lowest, average and highest ability levels. Testing against the lowest level ensures that a GUI or other such simplifying method is provided. Ideally, no console-driven or complex actions should be asked of this user level. Testing against the average level involves giving such users the choice of a GUI but should target the common tasks of the average computer user. Testing against the highest level provides the GUI system if needed but also supports greater solution or software depth with the understanding that this group will likely want full control over their environment. Using these three groups, it becomes possible to evaluate any conforming solution against a sample user population persona and enable developers to design according to these requirements.

3.2 Portability infrastructure

The aims stated in (3.1) can now be broken down further into a collection of minor points and examined within the context of workspace transference. Table 2 shows the breakdown of the aims discussed in (3.1) with the major components listed with the minor areas that make up each infrastructural point. The same requirements can also be seen in Figure 1, which displays how these aims fit together to form the framework.

Main requirement	Infrastructural requirements
Application portability	WorkTran application
	Native pass-though
	Emulation
Data portability and sharing	Data change detection
	Data synchronisation
Environmental settings	Environmental detection
	Environmental configuration
Interface requirements	User interface requirements
	Offline compatibility

Table 2. The workspace transference framework's major aims and sub-elements.

From the above set of aims, it is possible to breakdown the main workspace transference framework requirements into smaller and more specialised components. This section analyses these components and proposes a higher-level abstraction for the correct framework implementation.

3.2.1 Application portability

As specified in the framework aims for application portability (see 3.1.1), the framework should provide application portability between physical hosts. Figure 1 shows that this application portability layer is composed of three areas of functionality: the WorkTran prototype itself, natively executing applications and emulated applications.

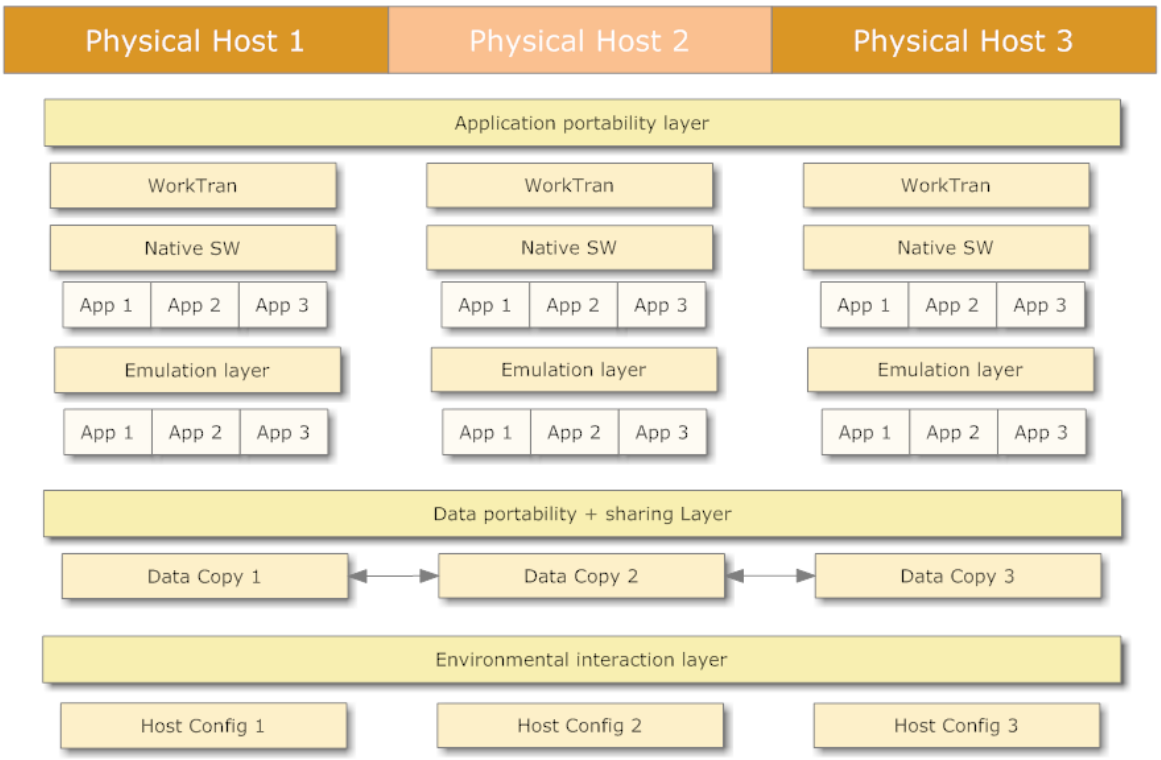


Figure 1. The workspace transference framework seperated into conceptual layers as it would operate across three physical host machines

The first aspect of the application portability layer, WorkTran, oversees application execution within the framework in the remaining two software layers. The WorkTran prototype should determine that an application is running within the portability environment and pass the executable object to the requisite layer based on its execution requirements. These requirements can range from simple pass-throughs (i.e. an application is already suitable for the same environment and hardware as the host machine and thus needs no additional emulation) or more complex interpretation in order to correctly counter the missing functionality. A consideration of the WorkTran prototype is that the program itself must be capable of running on most OS and hardware platforms. As such, the actual WorkTran prototype needs to be provided in a variety of forms for all the major OS and hardware combinations.

The remaining two layers support native applications: software that can run on the requisite hardware/OS combination unaided without any additional requirements, and emulated applications, which do require support. The emulation layer in particular allows any application compiled for a non-native platform to be easily transported between physical hosts. In this way, the WorkTran prototype can correctly allocate applications that can run on the native hardware platform to the localised host OS or pass the executable to the emulation layer fit for suitable mediation between the local host OS and the application OS.

A more detailed breakdown on the technical implementation of the conceptual design can be found in the Logic Model section of the appendix.

3.2.2 Data portability & sharing

The data portability and sharing framework requirements have been merged in the conceptual overview in Figure 1 since they represent the same architectural goals within the data transportation layer.

As shown in Table 2, this framework point comprises two areas: change detection and synchronisation. In the change detection stage, the prototype must be capable of detecting changes to any native or emulated data held within the prototype's environment by any application. Once data are manipulated in any way, the changes should, after an acceptable period, be synchronised with the peer nodes. These two infrastructure elements make up both the data portability requirement (see 3.1.2), since data are maintained on this and other peer nodes as needed, and the data sharing requirement, since it allows for multiple people to share data sets (see 3.1.3).

These two infrastructure considerations allow for data within the portable environment to be relatively fluid and represent, as closely as possible, changes from every node in that environment. While such data can be relatively easily synchronised between two peers, the data sharing aim (see 3.1.3) states that both the data and the portable application using it have to support dynamically refreshed data sources.

3.2.3 Environmental Settings

Since the WorkTran prototype is present as a software environment on each major hardware release (see 3.2.1), it provides a limited subset of host OS functionality to the contained applications. By aiding applications contained within the portability environment, the prototype can provide cross-platform functionality where required. For example, program functionality such as disk drive access should be transparently mapped onto the host OS functionality if required.

A further example of this functionality in action is a Windows application running within the Linux OS. This application should be capable of executing per the Application portability layer (see 3.2.1) within a suitable emulator that should provide a usable emulation of the Windows file system within the host system’s Linux OS.

In addition to executing the application, the prototype environment needs to import system-level information. A classic scenario from the familiar university environment is that of web proxies. These systems provide a standard method of caching data between the client OS and the Internet in order to reduce duplicate data requests. Unfortunately, most web proxies are non-transparent, meaning that the originating machine needs to rewrite its HTTP requests to establish an initial connection to the proxy server (which forwards the request onwards) rather than directly to the HTTP destination. Determining whether a connection should be proxied in this way depends on the physical host machine’s location within the network. Even though the proxy connection detail is a small piece of information, different OSs choose to store the configuration setting in a variety of ways. On Linux systems, this information is stored within a system variable, whereas on Windows systems, the setting is stored in the registry. WorkTran should be capable of extracting this

information regardless of host OS and presenting it in whatever method the contained applications require.

More advanced host OS functionality should also be provided as required and practical. One example of this is hardware access on the physical host. Access to locally provided devices, such as network shares or printers, should be provided within any application inside the portable environment as per the environmental interaction aim (see 3.1.4).

In order to provide this level of functionality, the prototype needs to be capable of detecting the physical host's OS configuration (Table 2). This detection phase can be followed by the pass-through to the portable applications in the configuration phase.

3.2.4 Interface Requirements

The interface requirement of the portability infrastructure is not present as a layer of operation within Figure 1 but instead represents the overall prototype usability goal.

As with the usability aims (see 3.1.5), the constructed prototype should be suitable for three main personas representing the technical ability average and extremes. Under this system, a set of criteria is constructed to test the prototype's success against a series of tasks generated for three personas in standard use scenarios.

In addition to the standard concerns of utility and ease of use, the prototype must also display the ability to work in a network-disconnected environment.

Computing environments lacking available Internet connections are increasingly rare. Nevertheless, this state remains possible. Offline compatibility ensures that a computing environment without Internet functionality is not at a disadvantage when used by any of the tested personas. Obviously, software specifically tailored to Internet connectivity (e.g. web browsers, email clients and other communication software) would be expected by the user to be non-functional in these situations, but other software (e.g. word processors and other general office software) should operate just as well without an active Internet connection. Likewise, data portability would suffer without an Internet connection since it would not be possible to communicate with peers in these situations, but the benefits of the framework should not be disregarded simply because of the lack of an active connection.

Offline compatibility is a necessity when a computer, for whatever reason, cannot be provided with connectivity during use of the prototype or any application within its portable environment. Security, airplane travel and other such environments exist in which a computing resource simply does not have an active Internet connection. In these situations, it is assumed that the host could have been physically moved (e.g. a laptop or netbook computer is taken away from a usable Internet connection), the connection could have been dropped for some reason relevant to the user or the workspace itself has been moved onto a different host (e.g. USB drive). In these cases, the workspace should provision as normal with no loss of functionality until the Internet connectivity is restored or the workspace is moved onto a machine with a usable connection.

3.3 Framework Summary

Existing research into workspace transference is extremely limited and no existing literature or technical development currently addresses the workspace framework requirements discussed in 3.1. The proposed framework attempts to address these gaps in future solutions.

The framework itself comprises the two sections discussed in this chapter, the aims (see 3.1) and the infrastructure (see 3.2) and details the further breakdown of each requirement.

The next chapter will discuss the creation of evaluation criteria to assess implementation of the framework within a potential prototype.

Chapter 4. Evaluation Criteria

To evaluate the outcome of the framework discussed in Chapter 3, it is necessary to select a suitable evaluation framework to assess any candidate projects created under it. In this chapter, the term *candidate* is used to define a project or software entity that is proposed under the workspace transference framework.

Should computer scientists experiment more[85] states that practical evaluation of a project is the best method of determining its success. However, the process of conducting an evaluation of any candidate is likely to be highly subjective[86].

Using the guidelines suggested in this paper, the following criteria were selected to evaluate a candidate solution:

- **Framework compliance:** the candidate's adherence to the workspace transference framework requirements as defined in Chapter 3
- **Persona use cases:** a testing methodology that mirrors the actions of a user via a series of use cases, including installation, day-to-day use and more advanced utilisation of the candidate software
- **Technical criteria:** a series of technical tests that detail extraneous points that are not visible from the user's perspective but are essential to the candidate's operation, such as synchronisation or higher-level functionality (e.g. efficient management of system resources)

In Section 4.1, a number of personas are selected from a pool of possible persona stereotypes and are then discussed along with the use case tests. This process involves discussion of each persona's traits followed by generation of an appropriate task list based on the perceived behaviour of each selected persona. After the use case scenarios have been selected, the more technical aspects of each candidate's design are analysed in 4.2 using a list of criteria generated for comparison against each solution.

While it would be desirable to also evaluate any developed prototype against other solutions, as the literature review discusses in section 2.4 there is no existing solution that bridges the gap to provide all the features of workspace transference as defined in 3.1. For example, while certain features of workspace transference can be provided by software in Table 1 no solution current provides all the requirements necessary for a even-sided comparison.

4.1 Persona use cases

The interface framework requirement (see 3.1.5) states that any constructed candidate should endeavour to provide a clear and concise UI. With this in mind, each candidate solution is assessed against the lowest, average and highest extremes of the user ability spectrum. From these three readings, it will be possible to view the solution from the various viewpoints of a proposed user base.

In *The Inmates Are Running the Asylum*[87], Cooper proposes a user-path model in order to evaluate the success of a piece of software. In contrast to the typical developer-based path taken though a piece of software, Cooper instead identifies personalities of a variety of potential users of the same software. With this method, the expected use of the candidate solution can be evaluated against a number of pre-defined criteria for each personality group. This model, to which Cooper refers as a *persona*[88], follows the simple form of an introduction of a stereotypical descriptive followed by a brief use expectation and a task list detailing the expected general habits to evaluate the system[89]. This persona model was chosen as the evaluation method for the candidate solution since it represents the clearest method of evaluation and interpretation of the overall success of any assessable solution.

Efficacy as defined in *Computer self-efficacy: Development of a Measure and Initial Test*[90] defines the concept by which an individual judges his or her own abilities against a number of factors such as outside encouragement, support, anxiety and outcome expectations. The measure of efficacy is useful in persona construction as it provides a defined model with which to assess an individual's capability with a specific piece of technology. A research subject with low computer efficacy can be said to have high task anxiety, that is, they feel that a given task is too complex for them to complete. In the context of these evaluation criteria, efficacy will be used as a basic measurement of task complexity against the persona's ability to perform computing tasks. In the persona generation shown below, the various traits associated with the efficacy scale are given collective names similar to the categorisation used in *The Impact of Users' Cognitive Style on Their Navigational Behaviors in Web Searching*[91], where different web-browsing techniques are personified to simplify the evaluation process.

From the concepts of persona construction and evaluation, three specific persona types have been selected to be applied to the candidate solution: Novice, Knowledge Worker and Developer, representing low, average and high computer efficacy, respectively. These personas allow for the evaluation of a candidate solution at the respective efficacy levels using an appropriate task list for each.

The use of personas was selected above use case actors as a persona can exhibit traits, in this case efficacy, as opposed to an 'everyman' approach where all actors in the system exhibit the same degree of knowledge.

The task lists for each persona are intended to demonstrate the following functionality:

- Package installation and maintenance
- Performance of typical tasks such as interacting with the files contained within the workspace
- Backup and recovery
- Workspace transfer across an OS boundary
- Demonstration of the above features within the new environment

This list covers the core functionality required by the workspace transference framework and forms the crux of the research outcome expected of the candidate solution.

For categorisation and ranking purposes, the term *efficacy* in the evaluation criteria refers to a computer user's competence and overall capability. A user with a low efficacy level would have rudimentary knowledge of computer use and very little technical expertise needed to diagnose hardware-related or medium to advanced software issues. Conversely, a user with a high efficacy level would typically fall under the classification of a 'power user' or 'developer'.

The three major persona groups are separated into sub-sections and listed in ascending efficacy level order below and a brief discussion of the rejected personas follows. For each persona, a description details the behaviour followed by a synopsis of the expected area of use for the candidate solution. The candidate solution is then evaluated against each of these personas along with a description of the process taken by each user in a step-by-step progression. A basic clean install of the Ubuntu Linux OS is assumed as the default environment since this was the primary programming platform. As specified in the application portability requirement, the candidate is required to function equally in any OS environment, including Linux, Windows and Mac. The screen shots in the below walkthroughs were taken from the Linux installation unless otherwise specified due to the convenience of the Ubuntu OS being available to the researcher.

These task lists are cumulative with each succeeding persona and incorporate the tasks of the previous persona. This prevents undue repetition while evaluating each persona. For example, the Novice persona installs a word processing package. This step is then omitted from the subsequent personas. This is not to say that the Knowledge Worker and Developer personas do not also need to install the same package; rather, the installation sequence has been omitted from those task lists to prevent unnecessary repetition of identical tasks. The Developer persona is exposed to higher-level tasks but this does not negate the possible use of the above word processor by this persona. With this in mind, each persona attempts a unique set of tasks. Avoiding such repetition, real-world users are expected to mix and match the persona types chosen here as needed.

4.1.1 Novice

The Novice persona represents users with the lowest computer efficacy. This persona typically expects technology to have low complexity and can become exasperated by any issues requiring technical knowledge. The ability to maintain a computer by keeping software updated is rare and the machine owned by this persona generally continues running out-of-date software due to the difficulty, perceived or otherwise, of making these updates as well as the lack of knowledge surrounding the upgrade process.

This persona is expected to be reluctant to initially adopt new technology unless recommended to do so by a close relative, friend or other word-of-mouth advice. After initial adoption, the ease with which software packages can be installed and automatically maintained and backed up is likely to be an advantage. Automatic software updates would generally be ignored by this persona for reasons

including apathy, general disinterest or lack of understanding. Upgrades would be still applied but largely go unnoticed unless a major addition to the package in question affects the UI.

The Novice persona stereotypically represents lesser computer efficacy and thus the tasks ascribed to this persona are not overly complicated compared to the subsequent persona types. The tasks below were generated by anticipating the everyday use of a desktop computer owned by a member of this group.

In the use case testing of each candidate solution, this persona performs the following tasks:

1. **Install the software on a desktop machine.** Intended to test the candidate solution viability when operating within a standard desktop environment. This task ensures that the installation sequence is simple and efficient for the Novice persona.
2. **Install a free virus checker package.** Ensures that installation of maintenance utilities such as virus scanners can be achieved without undue complication for this persona.
3. **Install a word processor package.** Similar test to ensure that installing more complex packages can be easily achieved by this persona. This package is also used in subsequent tests.
4. **Type and save a simple letter.** Represents general use of the environment with regard to a specific candidate-related action. This task also ensures successful operation of a word processing package, integration with the virus scanner, and successful implementation of the versioning system as the file is saved.
5. **Delete document contents.** This task is intended to cause intentional data loss of the original document created in the previous task to test the backup capabilities of the versioning system.
6. **Recover the document.** Retrieval of the changed document from task 5 from the server. This measures the effectiveness of the versioning system with regard to its ability to recover data.

These steps are intended to demonstrate the core aspect of the candidate using a simple scenario. In the above task list, the persona uses a portable application to generate, intentionally corrupt and recover a document. While the outlined scenario in tasks 3 to 6 is specific to a word processor-based document, the actual task list is also relevant to a number of similar tasks:

- Recovery of a deleted file
- Protection against data corruption (e.g. device failure)
- Remote data access. This is not expected to be as relevant to this persona but it represents a possible use of the candidate.

The above scenarios are incorporated in the Novice task list.

4.1.2 Knowledge Worker

The term Knowledge Worker, defined by leading business consultant Peter Drucker in the seminal *Concept of the corporation*, refers to a group of users such as students, lawyers, teachers, doctors or engineers who work with information in order to develop knowledge within the workplace[92]. Due

to the evolving nature of this role, the candidate will address this persona's portability requirements by providing multiple working environments including various office machines, home machines and portable environments such as laptops.

While the current trend for smaller and cheaper devices such as netbooks has increased within the workspace, the Knowledge Worker is still expected to use static workplace-specific resources such as desktop workstations. These static workspaces can include proprietary setups that must be contained within a specific working environment such as the constraints on the use of equipment attached to a dedicated workstation. Likewise, license-bound software for students or other professions can restrict the portability of such workspaces for legal or policy reasons.

The Knowledge Worker persona represents the middle-ground efficacy persona grouping within the persona-based evaluation between the lower-level Novice (see 4.1.1) and the higher-level Developer (see 4.1.3). The Knowledge Worker is expected to have a higher efficacy level when working with information and, therefore, be more familiar and comfortable with computers than the lower-level Novice persona but less able than the Developer persona. Since the work undertaken by this persona can vary, it is important that the environment this group works within is as adjustable as possible to address these concerns.

While the backup system is most valuable due to the relative perceived cost of the managed information, the packaging system could also be of use, quickly and efficiently providing transportable browser bookmarks and specialist applications as required.

The base workstations for a Knowledge Worker tend to be as varied as the groups' working methods but typically include laptops (prone to failure in older or cheaper models), USB disks or fixed hardware such as workplace-specific desktops or home machines. In each of these cases, the storage method is unreliable, underscoring the need for reliable data safety.

For testing purposes, the Knowledge Worker persona is asked to perform the following tasks typical of the workload of this persona:

1. **Install the software on a netbook machine.** Intended to test the viability of the solution with a device that has restricted resources. This point ensures that the installation sequence is simple and efficient for the Knowledge Worker persona and also ensures its compatibility with smaller screen resolutions.
2. **Install a web browser package.** This task demonstrates the ease and efficiency of each candidate's package installation. It also represents use of a package that needs to be continuously updated in accordance with security updates.
3. **Use web browser to bring up a web page.** This point is included to test the functionality of the installed web browser. It also demonstrates the transparent utilisation of local environmental settings such as networking options like proxy settings needed by the browser.
4. **Install the software on a second machine.** In this case, the desktop can represent a home or work desktop computer. The intent here is to show the synchronisation between the netbook and desktop machines. In this step, the account details used above are re-specified,

enabling the software to re-download all the above without needing to repeat the above tasks on the desktop.

5. **Open the web browser and view history.** Ideally, the web browser history should display the web page visited in the previous session on the netbook machine. This task demonstrates the successful synchronisation of a complex data set—in this case, the proprietary format storage for the web page history file.

4.1.3 Developer

The Developer persona represents a more varied, multi-skilled technical efficacy amongst the selected personas. This persona group has in-depth knowledge of software and a detailed knowledge of the host OS and software environments. Members of the Developer persona group can also have their own processes for data backup, synchronisation, versioning and all other features that the candidate provides, although most of these systems can be *ad hoc* in nature.

As a matter of preference, this group is more proficient with using command line interfaces and a standard GUI, partially because these interaction methods are easily scriptable and repeatable across a large number of hosts.

The following tasks are conducted for the Developer persona:

1. **Install the software on a desktop system.** Like above, this task forms a standard base for the candidate to run from and tests the efficient installation on the platform.
2. **Install a text editor and the Perl scripting language.** Together this editor and the interpreted programming language demonstrate the portability of the applications created by the developer. These two packages should demonstrate their cross-platform compatibility in later stages.
3. **Disconnect from the Internet.** This task demonstrates successful operation of the client without the need for an active Internet connection. This also tests the fault tolerance of the candidate in dealing with a sudden and unexpected network disconnection.
4. **Create and compile a simple program.** Using the GVim editor, a Perl script is created and is then converted into a native binary using the supported Perl compiler.
5. **Demonstrate the compiled program on the Windows platform.** This task demonstrates the portability of the application across an OS boundary. The application should show the same outputs on this OS in the same way as its own native compiled OS.
6. **Install the software on a headless remote Linux server.** Rather than relying on a GUI-based environment, this use case point measures the versatility of the candidate being able to function in a GUI-less environment such as a remote SSH server.
7. **Create a package from the application.** Demonstrates that the developer can contribute to the package sources on a different machine from the original with the source materials all being synced between the original machine, the Windows machine and this server machine. A package is created around the application along with prompts for relevant package information and relevant compatibility details.
8. **Upload the package to the server.** After the application has been packaged, this task demonstrates uploading of the package to the main server. This enables the package to be available to all other users (pending approval).

4.1.4 Rejected personas

The personas selected for evaluation were chosen on the basis of being the most likely audiences for a candidate workspace transference solution.

The following list details some persona groups that were eliminated during the analysis of possible personas:

- **Casual users:** By their nature, these users generally take the middle range of all persona groups, although they do share a resemblance with the typical Knowledge Worker persona. Like their similar persona group, however, this group can also have many varied working environments (laptops, desktops, USB disks), but unlike the Knowledge Worker, this persona tends to produce less actual data. Since this median persona can have varied levels of efficacy, the Casual user persona is similar to the Knowledge Worker but can also invoke elements of the Novice persona, while the more advanced members can even represent some forms of the Developer persona.
- **Roaming worker:** This persona refers to an individual who does not have his or her natural working environment but is allocated on a first come, first served basis. This is common in larger companies or amongst workers who for one reason or another are not allocated a permanent office or desk. An example of a member of this group would be a salesman who has to use any available resources as necessary without ever being allocated a set physical location. This persona is too close to that of the Knowledge Worker to merit its own use case trial.
- **Student:** This persona type is more of a clarifying subset of the above Roaming Worker classification. As above, the Student has no permanent workspace and uses university equipment to claim an available workspace.

These rejected personas are typically classified as mixtures of the three main personas and do not by themselves add any new diverse evaluation criteria to the persona testing stage. While it would be possible to extend the scope of this research by including more personas, the selected personas represent a broad cross-section without overcomplicating the testing stage. Extension of this research to include further personas is discussed as a possible future research area (see 8.2).

4.2 Technical criteria

Following the list of tasks generated for the selected personas in Section 4.1, it is also preferable to perform more objective technical tests on the candidate solution.

What follows is a list of evaluation criteria to test the more technical aspects of the candidate's design derived from the workspace transference framework requirement (see technical discussion in 3.2).

4.2.1 Distribution

This criterion discusses the distribution and initial set-up of the candidate and, more specifically, the packaging, set-up and initial impressions on the user. The distribution stage also includes the one-off set-up tasks such as initialisation of the program's environment (e.g. creating folders and installing default software) in order to fully install the candidate in order to set up the portable environment.

The distribution technical criterion requires that the candidate be distributed as a single executable file for ease of distribution in the form of a standard application installer. The candidate should ideally be represented as an executable file that can be run by the user to perform the initial set-up. This executable should have no external dependencies and be entirely self-contained and capable of installing without an Internet connection if it was obtained from some offline media (such as a USB stick or CD-ROM). If an Internet connection is present, this is obviously an advantage since the candidate can automatically download any updated software (including any new variants of itself), but any environment without access to the Internet must still be functional and accepting of available offline software versions.

Although this single candidate installation file can decompress and install into many other files or folders, the potential software user needs only obtain a single executable, the standard method of supplying installable applications for most OSs.

The executable must also be of manageable size, which represents a minimal download time on any modern Internet connection and supports an adequate range of OS standards, although in most cases this requires a separate compiled executable for each architecture (e.g. 32-bit and 64-bit executables).

The flow of a standard application installation sequence has been largely standardised and represents a set of necessary tasks. These are common for all applications installed within a standard Windows environment and represent the ideals of an installation sequence:

1. **Install core program files.** Needed to install the candidate and to set up the portable environment, which could be a directory on disk or on an attached physical device.
2. **Install shared libraries.** Not needed for the candidate since it will be a stand-alone application.
3. **Create and set OS-specific items.** The creation of OS-specific variables such as Windows registry keys is not needed since the single-machine limitations of these settings limit viability on multiple machines.
4. **Initially populate configuration files.** Relevant to the candidate in order to set up any initial packages and a portable environment.
5. **Create and set environment variables.** Not relevant due to the reasons specified in 3.
6. **Set up shortcuts.** Again, not relevant since the candidate should be portable between systems. Installing shortcuts on a specific machine may break functionality when any application executable (e.g. on a USB storage medium) is removed from that machine.

Through analysis of the standard installation stages, it is possible to determine the amended installation sequence for portable software:

1. **Select a destination.** The selection of a portable device or folder on the local machine onto which the candidate should be installed.
2. **Determine login details.** The optional selection/verification of a new or existing login to the online storage service is not immediately necessary but provides the credentials to perform online backups.

3. **Start the synchronisation.** An option to toggle the first synchronisation cycle immediately after the installation sequence has been completed.

The installation sequence should transfer control from the installation sequence into the now fully operational installation with the least possible intervention. This process should ideally have absolutely no interaction from the user and should be performed without the need for a system reboot.

Completion of this stage should include the set-up of a pristine portable environment in the location the user has selected. Unless the user selects to specifically not start the installed executable, the installation process should continue immediately into full operation by performing its first synchronisation (see 4.2.2).

4.2.2 Synchronisation

This section concerns tasks that are performed by the candidate on a regular basis such as data downloading or uploading, other networking concerns and the requirement for data versioning and the correct handling of network and storage media disconnections. These tasks are derived from the use cases outlined in the persona generation (see 4.1) and are not visible to the user but are instead employed as background processes during regular operation of the candidate.

The synchronisation task satisfies the data portability (see 3.1.2) and data sharing (see 3.1.3) requirements by providing a data transport layer between the candidate and its peers. Using this architecture, data can be transferred between any number of peers and be backed up on a central server.

Downloaded files must not be visible on the storage medium until they are completely downloaded by the candidate. The reasons behind this are expanded on further in the Package Management section (see 4.2.3), but this briefly equates to preventing partial package downloads. This process must occur transparently to the user without intervention except where requested. Failures must be handled gracefully and not require user intervention for resolution.

While network traffic is unavoidable during the synchronisation process, the network transfer process must ensure that communication between the client and server is minimal and not wasteful of available bandwidth. Data savings to and from the remote server can be negated by performance of a differential analysis over the local and remote files followed by compression of the generated differences. This process allows only the data set changes to be transmitted.

All client–server communication should ideally be compressed with the aid of a freely available and tested compression algorithm such as GZip[93], BZip2[94] or other data compression standard. These compression methods should be internally usable and avoid dependence on OS-shared libraries in order to maximise the candidate's portability. Most development environments provide the popular compression methods listed above and should endeavour to utilise them as much as possible during data transfers.

In addition to data compression, other factors affecting network speed or latency should also be minimised. Erroneous packet data is one such example that comprises data that, for various reasons, have not been received correctly by the destination and must be re-sent by the source. Since the candidate will send data in chunks and not necessarily limit itself to sending the full file, support should be given for the sending and resuming of a file at any time an Internet connection is present. For example, a large 10 MB file does not need to be sent from the client to the server all at once; rather, 5 MB can be sent at a time, followed by 2.5 MB and then the final 2.5 MB. At any point during this transfer, the user may change Internet connections, physical locations or environmental restrictions (such as firewalls). Under the standard one-shot file transfer method, this process would require that the entire transfer occur in one non-interruptible transaction. However, the candidate should be capable of splitting the transfer up into as many transfers as is needed.

The process to recover a previous version of a stored file must be relatively efficient and easy for the user. While typically the prerogative of the central synchronisation server, this functionality forms the data portability requirement of the workspace transference framework (see 3.1.2) and must be assessed from the perspective of the user with regard to the candidate's functionality.

This process links with the uploading process in which a file can be transferred from the client to server and the receiving server system reassembles the file, time stamps it and adds it to a version repository. Using this method, the server over time creates a full timeline of the file's life, from its initial creation to its eventual deletion.

Versioning itself is not the primary focus of this thesis; as such, it is not covered in great depth except for the functionality that allows the user to interact with the selected software installed on the server holding the relevant data.

4.2.3 Package management

The candidate manages applications within a native enclosure known as packages, which are a collection of resources that typically encompass an executable application. An example would be a web browser, word processor or any application that is prepared in a way that allows portability.

This criterion in turn analyses the handling of the package's life cycle within the portable environment. These stages are:

1. **Installation:** downloading of an initialisation of a package from the server.
2. **Maintenance:** automatic updating of a package to the newest version.
3. **Utilisation:** regular use and operation of the package within the portable environment.
4. **Removal:** removal of the package from within the portable environment.

The installation sequence for packaged applications must be efficient, easy to understand and involve as little user interaction as possible unless the user requests it. The installation sequence should also provide help that is appropriate to the user's efficacy level. Optional extras to the

package management process include providing suggestions that the user may find helpful for both use of the package and installation of similar or companion packages.

A package must never be left in a partial state and should only become available to the user when the application, its resource files and configuration are all completely installed within the portable environment. An example of this is the installation of any package where the application comes replete with multiple data files and configuration. During package downloading, it is desirable to not allow the user to be able to execute the application—even if the actual application executable itself is fully downloaded—because the application relies on its associated resource files for complete execution and throws an error upon finding these in a partial or corrupted state.

In order to work around this problem, packages are downloaded to a temporary location within the portable environment and then extracted and checked for integrity before finally being moved as a single transaction from the temporary location to the actual installation location. The outcome of this procedure is that the entire application, including all the required resources, becomes available immediately and in one atomic state rather than slowly as each file is downloaded.

This procedure is extended to upgradeable packages for which differentials of each file are downloaded to the same hidden location and once all are present (and the application associated with the package upgrade is not running), patching occurs as a single transaction.

Package maintenance will occur seamlessly as each updated package is downloaded from the server and undergoes an installation sequence similar to what was described above. The update process should ideally be completed with minimal user interaction so the whole process can be accomplished seamlessly. If package management is successfully implemented, then the whole package upgrade process should be accomplished without the user's knowledge.

Upgrade errors or requirements (e.g. backing up certain data) must be clearly expressed and provide help about how to resolve these issues.

Likewise, the removal process must be quickly and easily accomplished, help must be given where appropriate and full disclosure must be provided regarding what the package removal process involves. The consequences of any other affected packages must also be clearly outlined to the user. For example, if one package depends on another and both are about to be removed, the correct resolution to this conflict (both being removed) should be displayed for the user with an option to cancel the operation.

4.2.4 Error handling

The error handling criterion covers both network- and software-level disruptions with day-to-day use of the candidate software. Due to the portable nature of the environment, the actual program must be prepared for a certain level of unexpected disruption during its operation.

Sudden network disconnections must be handled correctly and the candidate should be able to gracefully work around any network disconnections and without data corruption.

This requirement essentially requires that the following two scenarios be handled correctly:

- Network traffic faults
- Physical storage device disconnection

Network traffic faults refer to an otherwise successfully established link to the upstream server, which, for reasons of routing, networking or other communication difficulties takes longer than a pre-defined maximum expected time. In modern OSs, the standard timeout for network communication is defined as 100 sec[95], but the value can be increased or decreased depending on the expected network latency of the user's location or Internet connection. While 100 sec is the maximum round-trip time for a user on a regular digital subscriber line (DSL)/asymmetric DSL/integrated services digital network, slower upload speeds could be present on other hardware solutions (e.g. satellite or dial-up connections).

Physical network disconnection occurs when the hardware is suddenly and unexpectedly removed from the system. Depending on the OS and network set-up, this hardware level disconnection can be handled by the OS in different ways. The most common way of dealing with such device disconnection at the OS level is to retain the device representation within the system but mark the hardware itself as inactive. This is easiest to handle within a programming language because although the networking traffic has stopped the device to which open handles are allocated, the socket libraries connected to the network device are not affected. This case is most common with physically wired Ethernet jacks since conceptually, even though the networking cable has been removed, the hardware that the device plugs into remains available to the system.

The second method of disconnection, which is seen with external USB, small computer system interface or some Wi-Fi technologies, is that of the entire hardware representation being removed from the system upon disconnection. This presents problems as open handles may be linked to the device (that no longer exists) that needs to be disconnected or released even though the devices they point to are no longer present. How this scenario is handled can vary from system to system, but the Linux kernel can generally cause programs to hang if the handle is not released correctly upon disconnection and some earlier Windows-based applications have been known to crash completely.

In any of the above disconnection scenarios, the candidate should offer a way to intelligently handle the disconnection without disturbing the overall operation.

The storage medium must also be handled as a resource that could be removed at any time. As in the above case of network disconnection, this can lead to a program with open handles to hardware that has been removed from the system. Again, depending on the OS and version, this can differ in the steps that need to be taken to gracefully handle removal of the storage medium.

Storage removal can also include the medium from which the executable is running. This can present problems if the executable expects resources at a given location on the now-removed medium (this behaviour is normal for the loading of pictures or icons needed by the application).

Most OSs attempt to prevent media removal by 'locking' the device until any open files or executing programs have released it. Unfortunately, while this mechanism prevents removal of the disk at the OS level (e.g. the user requesting removal of the media), there is nothing the OS can do to prevent physical removal of the device by the user while the files and executing programs are still using it. A typical example includes sudden removal of a USB storage device from which a program is running. Most executables, being loaded in RAM, handle this correctly, but any program that requires disk-stored resources (e.g. graphics, sounds and animations) usually raise errors when attempting to access resources that are no longer present for reading.

The candidate should correctly handle the removal of media and attempt to take steps to ensure that no dynamic reading of the media is needed past that of the initialisation, where the executable is loaded into RAM. This would prevent crashes upon media removal and enable correct recovery from these events.

4.2.5 Development environment

The development environment criterion is intended to measure the effectiveness of the candidate's development language with a view of the long-term viability of the project being sustained either by the original programmer or by a future programming team.

This criterion grouping is also relevant to the issues discussed in the Future Work section. 8.2 outlines a list of possible future projects as well as an assessment of these maintenance issues from the perspective of possible future developers.

While the ease of development within a programming environment may be subjective to the programmer, it is important to assess the rate at which development can commence.

In an attempt to assess this issue, the candidate's development is measured by its ability to be designed into fully operational software. This includes a recap of the implementation chapters for the candidate's implementation with a view of the complexities and obstacles involved with each iteration.

Reviewing the candidate development environment in this manner allows for assessment from the initial conceptual design through the candidate's development and the trade-offs made in each of the stages in order to achieve a working candidate. This process is expected to cover the development environment's use of common concepts such as object orientation, OS interaction (for device detection and environmental- and other hardware-related interactions) and the GUI system needed to present a graphical interface to the user.

After the initial candidate development stage is completed, it is necessary to allow for future amendments to each produced executable. Most development environments are created specifically with the assumption of a rigid and pre-defined specification leading to the final release. An equally important requirement, however, is the ability to confront changes should they become necessary

during development (as limitations are encountered with the development environment itself), with changing specifications or upon future development of a completed project.

While intrinsically subjective, this criterion attempts to assess the abilities of an average programmer to make regular bug fixes and small change requests during the implementation phase.

Although the core concepts of a development environment may be well established, the actual community effort that surrounds it is of equal importance in the expansion of the concepts that are sometimes omitted from the initial programming language's design. A classic example of this in action was the emergence of XML into already established languages. While most languages expanded on various core concepts by handling XML in object orientation like structures, the sheer dynamism of the language can cause huge problems for development environments that place limits on recursive declarations of limited stack space. Java, for example, was particularly affected since it prevents object nesting after a certain level of recursion, which can be a major limiting factor in its adoption of the XML language and its potentially unlimited nesting.

Most new concepts, such as XML and GUI systems, are usually first embraced as a proof-of-concept by the development environment community long before they are adopted by the core programming team.

The size, helpfulness and available resources of the community base of each development environment are measured with regard to the development environment as a whole. While not indicative of success, the community using a specific language can often contribute examples and documentation omitted by original developers for time or popularity.

The natural instinct here is to assume that older languages have better communities. While this may be true in the case of seminal bases like C/C++[96], corporate philosophy can also affect the success or failure of that individual community. A structure built around a business environment such as Java, for example, would be less likely to share code examples for fear of compromising internal business secrets; therefore, more open and hobbyist languages such as Perl or Python have a larger community with more readily available modules, tutorials and documentation.

4.3 Rejected evaluation criteria

In contrast to the selected evaluation described in the above sections, a number of other criteria were considered but eventually rejected from the final listing. The reasons for rejection include the criteria being either out of the scope of this research or insufficiently related to the workspace application framework goals. These rejected evaluation criteria are:

- **Advanced cross-platform testing:** While the aim of the application portability requirement of the workspace transference framework (see 3.1.1) is to perform adequate translation between different platforms, it is not possible to test for all permutations of the OS or other execution variables. This criterion was reduced in scope during the evaluation criteria by reducing the number of test platform combinations. This was due to the amount of effort required to test less popular packages on all possible OSs. In order to minimise the effect of this criterion, testing was constrained to the Windows and Linux environments, which account for 95% of current OS usage[97]. These two OSs account for the largest

distribution of platforms on the market today and encompass the largest cross-section of desktop, laptop and netbook setups.

- **Package-level testing:** This stage refers to the installation, use and removal of packages. While not necessarily an executable program, a package can represent any potential requirement on the system. Therefore, individual packages are downplayed in favour of the overall viability of a persona's use of the candidate. The reason for this is that most package contents are outside the scope of this project: each is independently maintained and distributed and varies in complexity and quality. Instead, the method of installation, maintenance and removal is tested, placing the responsibility for distribution on the candidate and not on external developers. The elimination of package agnostic testing consequently overlooks the contents of each individual package and focuses instead on package handling. While not considered a testing criterion, the suitability of the more popular packages (such as web browsers or other popular communications programs) would be accounted for on the Windows and Linux OSs within the candidate.
- **Internet connectivity:** From the perspective of basic Internet connectivity, the candidate should be compatible with the average environment expected of a regular computer user. Advanced networking options such as tunnels, virtual private networks (VPNs) or firewall rules are not tested. Instead, the candidate is assumed to be running on a regular desktop or laptop PC with moderate firewall rules such as the availability of standard web ports (HTTP port 80) and a working Internet connection.
- **UI compatibility:** While 4.1 addressed UI integration, the problem of UI compatibility using a multitude of different graphical considerations (e.g. screen size, colour palette, available resources) quickly makes this a very complex analytical point for all platforms. For example, a netbook PC has a much smaller screen size, which is known to cause problems with software created with the assumption of a larger vertical work area. This can mean that UI elements such as buttons placed at the bottom of taller windows can sometimes exhaust the available space on the screen and consequently become unusable without scrolling the screen or the user manually moving the window in order to fit the desktop area. While this is now being corrected in some applications, coding an application for these different display factors is still an issue for multi-platform applications. Some fixes can be incorporated into an application package, but ultimate responsibility lies with the original application designer to develop blueprints for compatible screen sizes.

The above rejected criteria were not used in the final evaluation criteria used to access the framework or the design stage in Chapter 5.

4.4 Evaluation Criteria Summary

In this chapter, two forms of evaluation criteria have been discussed: persona-based testing (see 4.1) and the more subjective technical criteria (see 4.2).

The Persona-Based Testing section discussed the construction of a number of personas to be used in the evaluation of the candidate. Of these personas, three were ultimately selected: Novice, Knowledge Worker and Developer (see 4.1.1 to 4.1.3, respectively). In each case, the candidate will be subjected to a variety of tests suited each persona's efficacy level. These tasks are intended to duplicate everyday usage of the candidate within the context of the persona tester and range from initial candidate installation to the more complex use of packages across multiple machines.

In the second part of the evaluation criteria, a series of technical criteria were generated to test the candidate's objective technical basis. These tests range from initial installation and use to regular candidate use.

In the next chapter, the above evaluation criteria along with the workspace transference framework requirements (see Chapter 3) will be used to design a prototype solution. This prototype will then be analysed against the factors discussed in this evaluation criteria chapter:

1. **Framework compliance:** Each of the framework requirements (discussed in Chapter 3) will be compared against the candidate.
2. **Persona tests:** The persona tests constructed in 4.1 are applied to the candidate.
3. **Technical criteria:** Additional technical criteria tests constructed in 4.2 are applied to the candidate.

Using these three evaluation criteria types, the design stage can now be performed using the prototyping method discussed in 4.1.

Chapter 5. Prototype Design

Following the generation of both a framework in Chapter 3 and the evaluation criteria with which to assess it in Chapter 4, it is now possible to implement the proposed candidate. This design process takes the candidate software solution from the previous evaluation criteria chapter and turns this theoretical entry into a constructible prototype software.

To accomplish this prototype design, this chapter identifies the key modules required for the construction to commence (see 5.1), maps out the overall system design (5.2) and then provides a workable technical implementation plan (5.4).

5.1 Framework integration

This section discusses the implementation requirements of the workspace transference framework (see Chapter 3) that dictate the prototype design.

Each of the workspace transference framework requirements will be examined and used to determine modules within the prototype design. These modules will then be used together in the next section (5.2) to create the prototype's technical design.

In ithis module, identification of the following factors will be addressed:

- **Overall prototype architecture:** Specification of the prototype from a high-level design standpoint.
- **Module identification and scope:** Identification of areas that are essential to the prototype creation process. Once these modules have been identified, their scope within the prototype is discussed and then analysed to answer the next point.
- **Module creation order and interdependency:** The order in which modules are developed within the prototype based on the above analysis.
- **Module communication:** Identification of which modules need to communicate with which others and how these communications channels are structured.

This section is organised into sub-headings in which 5.1.1 though 5.1.4 discuss the framework goals followed by a final design discussion in 5.1.5.

5.1.1 Application portability

As part of the application portability framework requirement (see 3.1.1), the prototype must provide support for applications to run on the host system regardless of the originally targeted platform. This section analyses how this can be achieved within the prototype as well as the aspects of the evaluation criteria against which the prototype will be assessed.

Section 3.2.1 discussed how the application portability infrastructure can be achieved in a prototype that implements the workspace transference framework. As shown in Figure 1, this can be achieved using a layered application framework, whereby a native application (i.e. a compatible host and application executable) can be executed by the host system in the regular way whereas a foreign

application (i.e. an application executable that was compiled for a different platform) can run within an emulator. This emulated environment will convert the application's internal OS logic into the native platform's equivalent and enable the application to run within the host system.

The prototype itself cannot be enclosed within a portable environment as it must oversee and manage the enclosed programs (i.e. the prototype can manage others but not itself). This restriction has been placed on the prototype in order for the prototype to provide various services to the enclosed programs, for which it requires operating system interaction.

The responsibilities of the prototype will be to transparently provide this level of application integration as well as a means to manage these applications. In providing such functionality, the prototype can gather an application into an atomic manageable unit (hereafter referred to as a *package*) that derives from the Linux definition of a container for updatable software[98]. Managing applications as these distinct interchangeable blocks allows for the prototype to effectively oversee each package's installation, upgrades and removal in a concise and user-friendly way. The inclusion of a Package Manager module would ensure that the prototype has a built-in method to manage applications distributed in this format and a means to manipulate these application sets as required.

In order to successfully satisfy the data application requirement, the following modules are proposed for the prototype's design:

- **Packaging:** A module that provides the packaging functionality detailed above. This module should be capable of installing, updating and removing any application package within the prototype's portable environment. The inclusion of a packaging system will satisfy the packaging (4.2.3) and application portability (3.1.1) technical criteria points within the evaluation criteria.
- **Emulation:** A module that provides functionality to run a foreign application within the native OS. The emulation system should ideally be integrated with the above Packaging module to provide a single method for updating the software as and when required. The Emulation module will also form part of satisfying the application portability framework requirement by providing the requisite low-level application support across multiple environments.

Further breakdown of these modules within the prototype can be found in the logic model design sections 5.2.1.1 and 5.2.1.4.

5.1.2 Data portability & sharing

This section of the prototype design merges two of the data portability (3.1.2) and data sharing (3.1.3) framework requirements. These two requirements are functionally similar during the prototype design stage in that the data portability requirement and the data sharing requirement share the same goal of moving data between prototype peers.

Conceptually, this combined framework goal requires that the prototype can be present on any number of computing environments that need to share data.

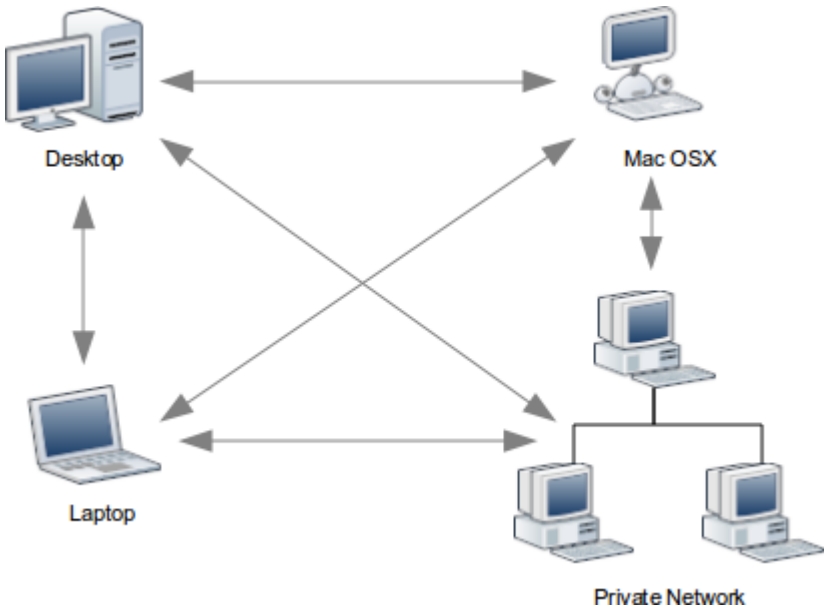


Figure 2. Peer-to-peer prototype networking model

Figure 2 shows an arrangement in which individual prototype nodes can intercommunicate and share data. While preferable from a logical standpoint, this method is impractical in real-world scenarios in which most PC systems exist behind a combination of firewalls and other security devices. It is not expected that this peer-to-peer arrangement would be practical from a portability standpoint, as all prototype peers would need to have a sustained connection to another regardless of network security policy. Such an arrangement would force all prototype instances to have an Internet presence.

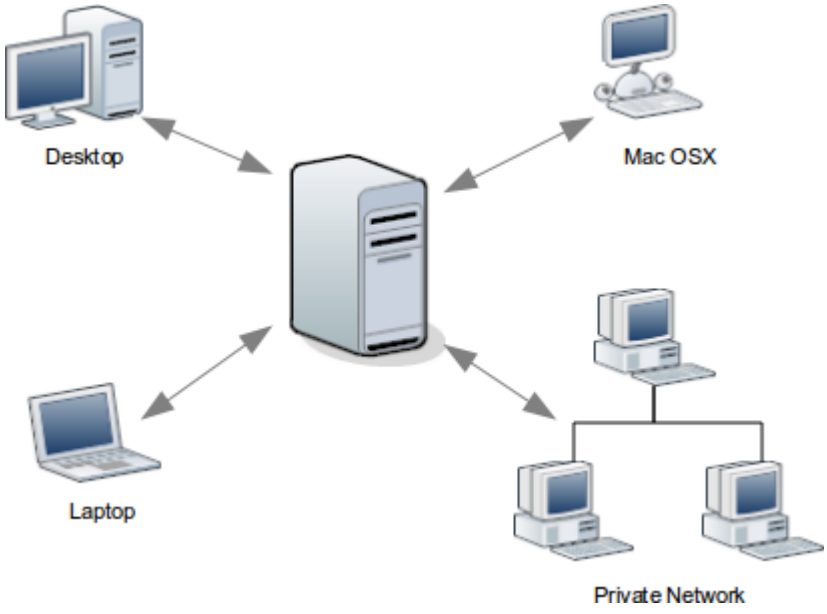


Figure 3. Network layout of centrally connected connected prototype nodes

Figure 3 displays a conceptual network of prototype peers linked though a central server. The central server (here using an Internet cloud hosting arrangement) allows the prototype nodes to communicate with any number of peers, the connections for which are routed though the server that is visible to all nodes. This method enforces only one intercommunication requirement, namely that

the peers need only be capable of communicating with a central server. This arrangement allows applications to be executed either on one host platform or multiple hosts assuming there are no data conflicts.

In order to successfully satisfy the data portability requirement within this sub-section, the following modules are proposed for the prototype's design:

- **Networking:** Provision of networking access with which the prototype can communicate with the next module. By wrapping the cross-platform functionality of the host environment's networking, inclusion of this module satisfies the synchronisation (4.2.2) and error handling (4.2.4) technical evaluation criteria.
- **Central server:** A centrally provided server with which the individual prototype nodes can communicate. Inclusion of this module satisfies the synchronisation evaluation criterion (4.2.2) by providing support for a remote server and versioned file recovery on the server side.
- **Synchronisation:** A module intended to compute the differences between data blocks and transmit these computed differences to the central server shown above. This module addresses its namesake evaluation criterion of synchronisation (4.2.2) and provides support for the packaging system (4.2.3).

Further breakdown of these modules within the prototype can be found in the logical model design sections 5.2.1.7, 5.2.2 and 5.2.1.5.

5.1.3 Environmental interaction

The environmental interaction framework goal discusses the interaction between the host OS and the hardware platform. This section discusses the factors affecting the prototype when it is run within these differing environments.

Owing to the inherent difficulties in maintaining support across multiple platforms, it is necessary to supply various mechanisms by which to access host level functionality. While some programming environments already provide cross-platform functionality in this way, these supplied methods may still require specially created helpers in order to arbitrate specific OSs.

An example of the need for such cross-platform compatibility can be seen in the way that different OSs address file and directory path management. Table 3 illustrates the basic stand-alone location of a user data storage directory on the different OSs and the obvious differences in file systems among multiple OSs.

OS	Path
Windows 95	C:\My Documents
Windows XP	C:\Documents and Settings\USER\My Documents
Windows Vista	C:\Users\USER\Documents
Windows 7	C:\Users\USER\My Documents
Mac OS 9	Users::USER::Documents
Mac OS X	/users/USER/Documents
Linux	/home/USER/Documents

Table 3. Comparison of different operating system path formats when addressing the users personal storage directory

The problem illustrated in Table 3 is present in the prototype when determining the user's data storage directory on various OSs. Each OS specifies guidelines for how this location should be determined. For Linux, this is determined via the 'HOME' environment variable (although this can change with some Linux flavours) and usually follows the '/home' path as specified in the Linux File-system Hierarchy Standard[99]. Apple provides a strict design document detailing its user storage standards[100], while Microsoft provides a number of API calls to likewise determine the user's storage location.

As shown above, the pathing scheme changes radically between Windows versions, and Microsoft tends to also change its API, which sometimes results in confusing and/or contradictory functionality. Even in the case of the above path examples, Microsoft seems to switch between MSDN published documentation and various 'unofficial' (albeit still Microsoft supported) articles to clarify some of the Microsoft Windows APIs and determine the location of the storage directory (Table 3). The most widely used of these is Chen's online blog, *The Old New Thing*[101], where some of the more complex or subtle nuances can be clarified for Windows programmers[102].

With the above differences, it quickly becomes apparent that each OS presents its own unique path to the common concept of document storage. This is just one of the more visible areas in which the prototype will find problems providing a cross-platform implementation in order to satisfy the data portability (the data being stored in various disk locations) and environmental interaction (the cross-platform way in which the data are stored) goals. The prototype needs to account for these changes and attempt to unify these differences across all environments, solving the above confusion on behalf of the user and to the satisfaction of the framework. The solutions presented in the prototype (such as path resolution) should be supported in any program contained within the prototypes portability environment.

In order to successfully provide environmental interaction, the following modules are proposed in the prototype design:

- **File system access:** A wrapper module that corrects host system-specific functionality such as file paths (as per the above example), writing, reading and other functionality that will need to be correctly handled by the prototype regardless of the target OS. Since these features are largely supplied as utility functionality, this module does not actually represent a given evaluation criterion except for a possible discussion of the prototype development environment (4.2.5).

- **UI support:** Support for addressing and providing feedback to the user. Since the prototype is intended to be cross-platform compatible, the UI is expected to require some support for its cross-platform implementation. This module is covered in further detail in 5.1.4.

Further breakdown of these modules within the prototype can be found in the logic model design sections 5.2.1.6, 5.2.1.2 and 5.2.1.3. The UI is discussed further in 5.1.4.

5.1.4 Interface requirements

The prototype must satisfy the interface requirement by offering some means of user interaction. While this task in itself is not difficult to achieve with an application aimed at one OS, the selection of compatible UI technologies is limited by the cross-platform requirement of the framework. The prototype UI must run natively on the host OS and no show functional differences between OSs as stipulated. While some differences are inevitable (e.g. moving various UI widgets to accommodate the host screen resolution), the actual difference between OSs should reflect the application portability requirement, which states that applications should run within other OSs without difficulty.

To achieve this requirement, the UI system should be constructed in such a way as to minimise the effort required during development time in testing cross-platform toolkits.

To ensure efficient debugging of the prototype during the implementation phase, the UI development is split into two groups: first, a command line interface (CLI) was developed since this interface represents the simplest method of human interaction via a text based prompt system and aids in the debugging process. Next the more complex cross-platform GUI system is created. During the GUI construction stage the selection of toolkit is vital to ensuring that the prototype can operate in a cross-platform manner.

From this analysis we can determine that the following modules are required during the prototype design:

- **CLI:** Primarily used for debugging, the CLI represents user interaction at its simplest level. This module, as with the following, satisfies the all the user-facing evaluation criteria of the prototype including initial distribution and the UIs of the packaging tasks (see 4.2.3).
- **GUI:** Represents the main method of user interaction with the prototype by consistently displaying the prototype's UI across platforms. As with the above CLI stage, this module supports the above evaluation criteria points.

Further breakdown of these modules within the prototype can be found in the logic model design sections 5.2.1.2 and 5.2.1.3.

5.1.5 Framework integration summary

The above sub-sections have analysed the workspace transference framework requirements and suggested a number of modules that must be integrated into the prototype design.

Framework Requirement	Implementation Modules	Evaluation Criteria(s)	Further Discussion
Application portability	Package manager	Package management (4.2.3)	5.2.1.1
	Emulation	Application portability (3.1.1)	5.2.1.4
Data portability and sharing	Networking	Error handling (4.2.4)	5.2.1.7
	Central server	Synchronisation (4.2.2)	5.2.2
	Synchronisation	Synchronisation (4.2.2)	5.2.1.5
Environmental interaction	File system access	Application portability (3.1.1)	5.2.1.6
Interface requirements	CLI	Application portability (3.1.1)	5.2.1.2
	GUI	Application portability (3.1.1)	5.2.1.3

Table 4. Outline of required modules during the prototype design stage

Table 4 shows the list of required modules determined using the above framework requirement sections. This table illustrates each framework requirement along with each outcome module and the evaluation criteria its inclusion would satisfy. The section that discusses each module in further detail is listed in the final column.

The next section will discuss these extracted modules in relation to the prototype's design and implementation stages.

5.2 Logical Architecture

Now that the prototype's key modules have been identified in 5.1, it is possible to generate the prototype's initial conceptual architecture.

This section represents the higher-level prototype design overview by which the functional modules within the prototype are detailed and their responsibilities and inter-relationships are analysed. These are drawn from the discussion in 5.1 and reflect the modules shown in Table 4.

The decision to employ a client–server architecture was made in order to work around the issues with peer intercommunication discussed in 5.1.2. These issues centre around the problem of peers being blocked from directed peer-to-peer communication due to network restrictions. Thus, a central point of communication must be employed in the architecture to mediate the data synchronisation and application portability. Peers can communicate with this central point as needed to exchange information and satisfy the synchronisation technical criterion (see 4.2.2).

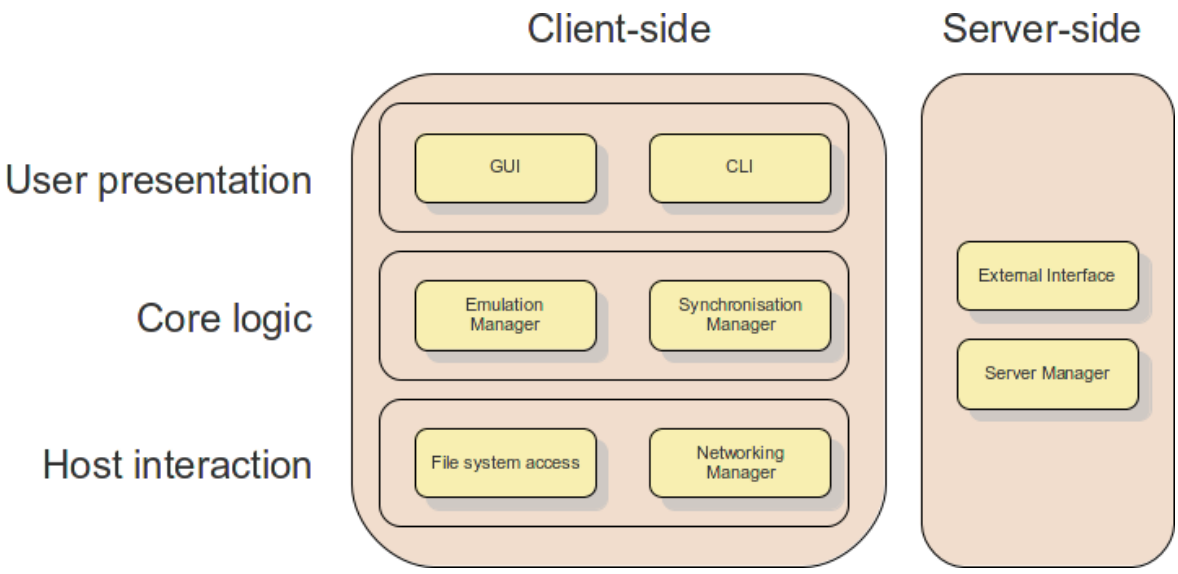


Figure 4. Logical overview diagram of the prototype architecture

The system design shown in Figure 4 is derived from the key modules identified in 5.1. These modules fall within two major deployments: client (i.e. the individual prototype end-points within each deployed portability environment) and server (the central mediation server). These modules are also classified into four main categories:

- **User presentation:** The modules responsible for the prototype's interaction with the user. These comprise both a simple CLI and a more complex GUI. See sections 5.2.1.2 and 5.2.1.3 for more information.
- **Core logic:** The modules responsible for the central logic and prototype purpose. These two tasks, namely emulation and synchronisation, support the workspace transference framework requirements within the prototype and provide most of its functionality. Further discussion about these modules can be found in 5.2.1.4 and 5.2.1.5.
- **Host interaction:** These modules provide the prototype's low-level utility functionality. Specifically, they interact with the local host environment in a cross-platform–compatible way to provide a consistent interface between any targeted platform. Further discussion can be found in 5.2.1.7 and 5.2.1.6.
- **Server side:** This grouping of modules, apart from the main client prototype instances, provide the prototype's central server functionality. The modules here are designed to provide an interface to each connecting client prototype node and its own server-specific functionality.

A description on each of the individual modules and their functions, inter-relationships and child modules can be seen in sections 5.2.1.2 though to 5.2.2, respectively.

5.2.1 Client-Side Components

This section discusses the major units shown within the prototype overview (Figure 4) that are specifically related to client-side operation. As the data sharing requirement of the workspace transference framework states, any prototype should be designed with the intention of supporting multiple instances of the prototype on multiple machines. This is intended to allow a user to manipulate synchronised data simultaneously from any machine. The description of each module

lists the tasks that must be performed on each local copy of the prototype in communication with a central server, which in turn distributes the shared information between nodes.

An example of this requirement in use would be Machine A running one prototype and Machine B running another and either machine can be running any combination of software or hardware. In each of these cases, machines A and B would possess a duplicate of the prototype running on each native OS and the modules described below are all expected to function the same way regardless of local restrictions or configuration.

5.2.1.1 Package manager

The Package Manager module represents the prototype's application management functionality discussed in 5.1.1.

An application contained within the prototype portable environment is referred to as a *package*. This term is used to illustrate that files or data contained within this designation should be moved as a single transferable unit when transported from or to the remote server. The term *package* is extended in the prototype to also potentially contain any amount of data that requires atomic synchronisation as a group between the client and server.

In this atomic synchronisation process, any contained file is effectively grouped together where any single item is not allowed to change state (i.e. be updated, moved or deleted) without the remaining group members also doing so. This is most typical of software for which an update needs to be simultaneously applied to the application, its data files and any extraneous files associated with it. Failure to maintain this transaction-based atomicity means that the actual application may be a different version than those of the resource files or configurations. Therefore, any collection has an enforced all-or-nothing approach to synchronisation in which each data store is referred to as a package.

All of the package-based tasks occur via updating of the internal state of the remote server and then invoking the local prototype's synchronisation manager functionality to pull the remote changes to the client. This method was chosen as being the most viable due to the prototype's requirements of being active on multiple machines simultaneously. This means that each machine can immediately retrieve the changes rather than one client performing the action, pushing the information to the server and relying on the sibling prototypes to invoke the synchronisation process.

During installation of a portable package within the portable environment, the file transfer occurs and the package is then unpacked with any pending post-install scripts to finalise the process.

Upgrading involves the same operation as the install process with the optional stage of merging and transporting user settings between the two package versions. This allows packages to retain settings from previous package versions.

Removal of packages, as the name implies, simply removes all of the package files from the local environment, the removal of which is signalled to the synchroniser, which duplicates these changes

on the server and all other nodes. The removal process is effectively the same as the user deleting the package directory, an action that would be duplicated to all other peers in the standard way discussed in 5.2.1.5.

5.2.1.2 Command line interface

As shown in Figure 4, the aspects of the UI are separated into two modules: GUI and CLI. This is not an explicit requirement drawn from the workspace transference framework; rather, it is necessary to provide a suitably simple interface from which to debug the prototype's implementation phase. The two separate interfaces are specified in order to provide two distinct methods of user interaction during the development process. Since creation of a CLI-based program is much easier, a similar CLI UI can be created first and the GUI interface components later. Likewise, the CLI-based system requires minimal maintenance and can be debugged during the implementation phase with less effort than a full GUI environment.

CLI systems generally take one of two forms:

- Zero interaction shell-driven command systems
- Ask-response–based systems

The first method simply consists of wrapping the functionality of an application around the initial invocation of the program to which the application then responds and exits. The second supplements this basic functionality by interactively asking the user a series of questions. Both of these methods are relatively easy to implement and involve no prerequisite knowledge of GUI-based systems or any user interaction except for sanitising and processing user input in the second case.

The CLI module represents the simplest method of controlling the prototype via the regular CLI. This is intended to act as a scaffold to allow for the development of the core research concept without the hindrance of a GUI-based system early in the development stage.

Since the main components of the prototype are non-graphical, this interface is intended to act as a straightforward, scriptable CLI system to control the prototype's environment in the simplest manner. By breaking down the user input in this way, the core functionality can be perfected first with the more complex GUI and user interaction stages occurring after the essential application functionality has been completed.

Using only the CLI-based system, it should be possible to invoke all of the main functionality outlined in the workspace transference framework. This should include package installation, removal and information gathering, synchronisation with the server and interaction with installed packages and the various cross-platform emulation layers.

5.2.1.3 GUI

The GUI module, in contrast to its much simpler CLI sibling discussed in 5.2.1.2, is intended for widespread use and is aimed at all levels of the user technical ability spectrum. Its inclusion within

the prototype is stipulated in the UI workspace framework requirements (see 3.1.5) as well as the persona tests for the lower efficacy Novice persona (see 4.1.1).

Each aspect of the GUI module is actually a separate windowing system that is capable of offering various pieces of visual feedback on a prototype's options.

Additional coding concerns, such as maintaining this GUI system module, should be left until the prototype's core functionality has been fully developed and tested against the much simpler CLI module.

5.2.1.4 Emulation manager

The emulator module provides the application portability functionality within the prototype.

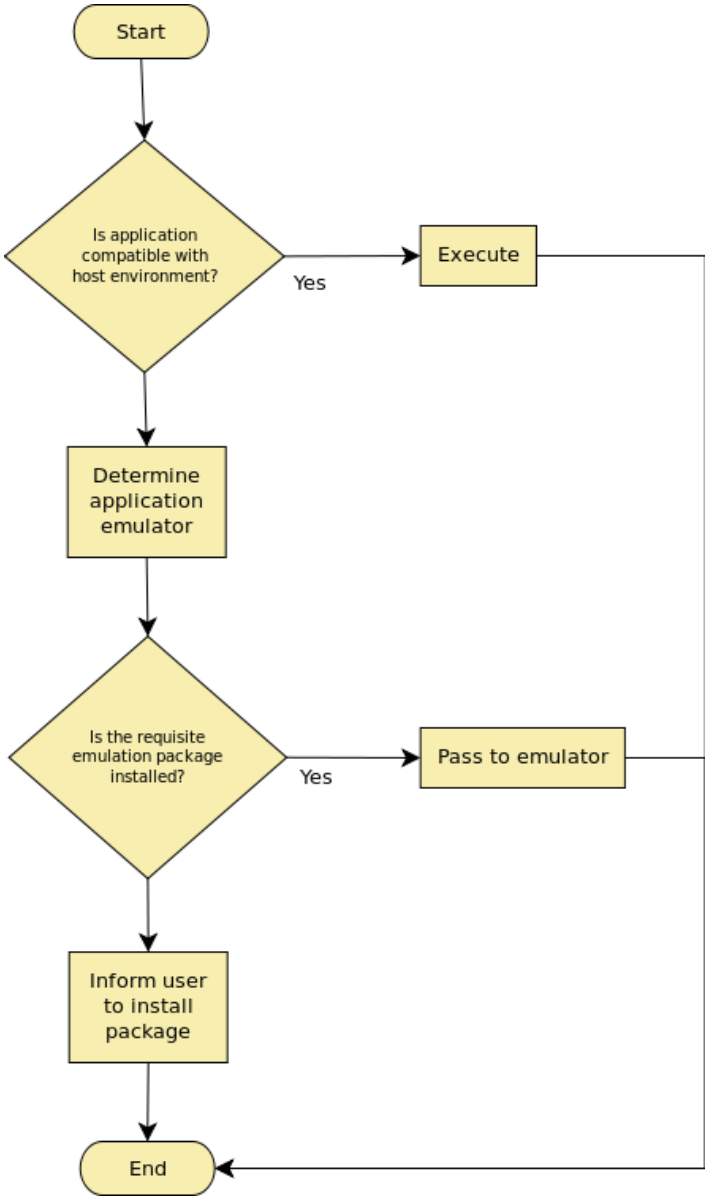


Figure 5. Emulation logic model

As shown in Figure 5, when an application is executed, the prototype will determine if the application is native to the local host environment. Should this be the case, the application is launched in the regular way. If the application is not compatible with the host environment, the prototype first determines which emulation platform is required to launch the application and then passes control to the determined emulator. A minor caveat to the process is that if the requisite emulator is not already available, the user will be prompted to install the appropriate package.

Using this approximate model, the constructed prototype will be able to run both natively compiled applications and applications that were originally designed for another architecture.

5.2.1.5 Synchronisation manager

The Synchronisation Manager module is primarily involved with the control of the external Unison system, which computes the data that needs to be transferred to the server.

The synchronisation manager itself is responsible for managing various configuration rules such as what files should be kept identical between the server and the active environment, which files should be ignored and what the priorities are for each file type. These settings are then translated into Unison's own profile format and executed via the inter-process communication (IPC) system. The actual execution process interaction occurs via the Predict module, which manages the Unison sub-process during the synchronisation sequence (see 6.3.1 for further details about the Predict module).

5.2.1.6 File system access

As discussed in 5.1.3, compatibility with the host environment is not always handled at the programming environment level. File system paths and accompanying calls will need to be rewritten by the prototype depending on the currently active host environment.

The File System Access module exists for this purpose, whereby a common interface can be presented for the prototype’s use that can internally map the relevant low-level file system functionality.

The prototype programming environment selected later in 6.1 should ideally include a certain amount of host OS compatibility and should reduce the need for translation of the prototype’s implementation. The inclusion of this module can therefore be seen as a wrapper module, where the requested functionality can be translated into the relevant host operating environment equivalents.

5.2.1.7 Networking manager

The Networking Manager module handles communication with the server-side external interfaces (see 5.2.2.1) in that it provides a consistent method for accessing the low-level networking functionality of the host environment.

While common local tasks such as file management have been integrated into the core functionality of most languages, most of the more popular programming languages in use today predate the

widespread use of the Internet or at least some of the more common protocols such as HTTP[103]. Because of this, most programming languages attempt one of many different methods to integrate Internet-based transports into their core language:

- **File system integration:** This is simply an extension of file system functionality that allows the existing file system to access operations such as querying, navigating, reading and writing, all of which are transparently mapped to their networking counterparts. This is used to the best effect in file wrappers such as those in the scripting language PHP[104], which provides a stream API[105] for pluggable integration of any potential concept into the medium of regular file system access.
- **Push/pull API:** This is usually a simplified version of the socket call method below. A file or piece of data is provided and the networking API layer executes the appropriate internal socket connections. This is a lower level of the abstraction than the above file wrapper method and forms the most popular method of Internet transport.
- **Socket API:** The lowest common denominator of communication, sockets are provided at the OS level with the programming language usually providing a thin transaction layer between the executing code and the lower-level OS interaction. This layer can increase in complexity if the language in question is of a cross-platform nature, as is Perl and Python, in order to resolve the inconsistencies of the underlying OS. This can mean that higher layers need to perform arbitration between the different OS socket access methods.

Each of the above functionality points indicates a progressively lower level of abstraction, thus the socket API represents the lowest and most technical basis for file transfers, while file system integration represents the highest and most abstract functionality. Ideally the second method of the push/pull API should be implemented within the prototype as it provides the best convenience when coding the higher-level logic whilst removing the underlying communication complexity.

5.2.2 Server-Side Components

As with the client-side components, the prototype requires a number of server modules to efficiently distribute various actions across multiple environments. Most package management operations are performed on the server in order to centralise operations in a trackable and snapshot-compatible way. These snapshots are intended to allow the user access to past versions of the same data set, a concept that is synonymous with 'rewinding' a file to a point in its own history. While perfectly feasible technologically, the disk space and resources required to implement this on the client side are usually more demanding than the storage space on the prototype's installed volume (typically a regular USB drive) and more demanding for calculating the computational differentials using the client-side CPU and RAM resources. For this purpose, the versioning information is kept on the server, which has much larger data stores and resources to maintain these data stores. Using this method, the client can upload any amount of data and the server can sort, categorise and correctly version the data internally. The reasons behind this is to prevent data corruption either by the user (e.g. by overwriting *good* data) or by any contained portable program (e.g. a fault in the program). An example of data versioning and the reasons behind its implementation can be seen in the scenario outlined in 6.4.1.5.

5.2.2.1 External Interfaces

The external interface acts as a variety of available services running on the server. All of the services support various methods for accessing files, packages or user settings within the server though various APIs.

In principle, the contents of the server's file storage should match the contents of each prototype environment. This rule has exceptions when files or directories are specifically ignored or prioritized for each environmental setup.

Due to their tried and tested simplicity, the standard HTTP, RSync, SSH and FTP protocols were chosen as communication methods with the server. These four give a wide variety of scriptable API access methods to the server storage.

Additionally a XML API Web Interface is exposed to allow prototype communication between client-server and client-peers. This presents a unified method for client-server communication and allows optimized file storage allowing for time-based differentials including the rewinding, merging and versioning of file contents from any snapshot compatible point within each interface's history.

The XML API is also capable of operating as an interface to both the prototype and to any other communicating program should future extendibility be required later. Future development examples are more concisely listed in the conclusion and future research chapter (see 8.2) but could include examples such as external AJAX or other Web 2.0 based systems from third-parties.

5.2.2.2 Server Manager

In addition to standard file access on the server via the above protocols, a number of modules must be created to allow for efficient management of the subscriber's files and packages. These can be either simple maintenance programs that optimise disk space in order to maintain server storage or more complex file-system functionality that maintains versioning data.

The server manager is responsible for appropriately maintaining the linkages between packages on the server that are ultimately downloaded onto each client environment by the Unison program running on each client.

The server's duties are largely related to package and stored file version management. Its single centralised nature makes this the easiest component to manage, both by package maintainers and by prototype designers.

5.3 Component interaction

This section outlines a number of in-depth case studies in which each component of the prototype is shown to perform its task and describes how these modules are inter-related. While not indicative of *every* possible use of each prototype, these scenarios are intended to list a number of test scenarios and provide in-context examples of how each component module should function in its completed state.

The following cases were selected as representations of the main prototype use. These cases involve package installation, regular use and installation/removal. These actions constitute the prototype's core functionality and show the interaction between the modules as a general overview of inter-module interaction.

Module names are highlighted in bold for clarity.

5.3.1 Prototype installation

A user initially installs the prototype onto a computer within a prototype-compatible environment. When first running the software, the user is presented with an installation screen by the **GUI** module, which asks for the location on the portable environment where the software should be installed. This destination typically corresponds with a removable media device such as a USB stick, but it could also apply to any storage medium including network drives or a local hard disk.

The software is copied into the prototype environment and populated with minimal information such as a default prototype-relevant configuration. Such examples may include a basic web browser or other 'start-up' software. These pieces of software are simply example packages that are installed so the user can get a feel for how the prototype operates.

5.3.2 Synchronisation

Should the user create, edit, delete or perform any action upon a file or setting within the prototype's managed workspace, this operation is detected by the **Synchronisation Manager** and the relevant files are marked as altered. After a given delay, the **Synchronisation Manager** prepares the differentials on these marked files and uploads the changes from the local file system to the server via the server's **External Interface**.

The **Synchronisation Manager** (in an attempt to remain as low profile as possible so as not to disrupt the user) intelligently monitors RAM and CPU resources before uploading a changed differential. CPU monitoring (referred to as 'load' on Linux platforms) can be employed to prevent undue slowdown during the upload operation. Network bandwidth is also monitored to prevent any noticeable slowdown for other applications such as web browsers. If the CPU load is low enough and no subsequent activity is detected, the upload interval should be approximately 15 minutes. This changes by file type, as some (document, spreadsheet and plain text files by default) have a smaller update synchronisation interval than other typically larger files. In reality, this interval changes based on the system resources, but ideally every file should be uploaded within one hour of the contents being detected as changed.

Factors such as the available CPU load, RAM resources, bandwidth and percentage of files changing before an upload occurs as specified according to file-type rules or are subjected to a global rule if no matching file type is determined. All of these settings are alterable by the user within the preferences panel of the prototype.

5.3.3 Package installation

When the user requests package installation either though the **CLI** or the **GUI**, the relevant interface requests that the installer sub-module of the **Package Manager** perform the installation. The installer sub-module forms a request to the dispatcher sibling, which sends the request off though to the server-side **External Interface**. The **External Interface's** XML API then dispatches the request to the **Server Managers** linker, which checks the file system environment and symbolically links the package to be installed inside the file area.

On the next synchronisation cycle, the **Synchronisation Manager** processes the change and begins to download the package. When completed, control is passed asynchronously to the **Package Manager** installer, which finalises this process and runs any remaining installation and configuration scripts.

Finally, the **Package Manager** informs the main interface (either **GUI** or **CLI**) that the install is complete and that the user should be notified.

5.3.4 Package upgrade

A user requests a package upgrade through either the **CLI** or the **GUI**. Either interface passes the command onto the installer sub-process of the **Package Manager** module. The installer is used because an upgrade is treated internally like an installation except at the very last stage.

As with the installation screen, the installer child communicates with its dispatcher sibling that a message should be sent to the server's **External Interface** via the XML API. The **External Interface** XML API then communicates with the linker child of the**Server Manager** to install the package.

At this point, the linker recognises that the package is already installed. The Linker sub-module links the requested package upgrade instead into another non-conflicting directory.

On the next synchronisation cycle, the **Synchronisation Manager** detects the changes and conflicts between the two installations. All files in the package are checked for use by the OS. If any file is detected as being viewed or locked, the upgrade sequence is deferred until the next synchronisation cycle.

When the files become available, the two active package versions are merged, effectively upgrading the older package version.

5.3.5 Package removal

The user requests that a package be removed via either the **CLI** or the **GUI**. The relevant UI informs the Destroyer module of the**Package Manager** that an uninstall should take place. The child runs any uninstall scripts on the local installation and then passes a message to the dispatcher sibling.

The dispatcher child then communicates the uninstall request to the **External Interface's** API interface, passes the uninstall prompt to the **Server Manager's** linker child, which removes any server-side reference to the software.

The scope of the prototype construction makes it necessary to generate an objective method that allows each iteration of the implementation stage to be judged as either complete or incomplete.

5.4 Prototype design summary

This chapter has analysed the modules essential for the construction of a workable prototype to demonstrate the workspace transference framework. In this analysis, it was discovered that a workable structure can be created based on a client–server architecture between the prototype's environment and a central server, which can in turn redistribute the data to any other connected prototype peer nodes.

Using this method, the prototype can satisfy the main requirements of the workspace transference framework in the following ways:

- **Application portability:** Provided in the Synchronisation module when transferring a package's environment between nodes.
- **Data portability:** The Synchronisation module provides a dependable and stable communication interface between the client and the server. Using computed data differentials, the Synchronisation and Networking Functionality modules can maintain a set of files between the client and the server.
- **Data sharing:** Similar to data portability, the data sharing requirement is satisfied by the Synchronisation and Networking modules. Using the networking architecture discussed in 5.2.1.7, it is possible to distribute data between any number of prototype nodes.
- **Environmental interaction:** This requirement is satisfied by the use of a cross-platform file system manager as described in 5.2.1.6 beside a cross-platform module as described in 5.2.1.4.
- **Interface requirements:** The prototype provides both a CLI (5.2.1.2) and a more complex cross-platform GUI system (5.2.1.3).

Following this design phase, prototype implementation can begin along with assessment according to the criteria constructed in the evaluation criteria chapter.

Chapter 6. Initial Prototype

This chapter details the construction of a prototype to demonstrate the practicality of the workspace transference framework described in Chapter 3. The prototype described in this implementation chapter is constructed according to the design proposed in Chapter 5 and then evaluated according to the criteria constructed in Chapter 4. During this implementation phase, the prototype will first be constructed according to the high-level design described in Chapter 5 as well as the lower-level technical design discussed in 1.2. After this construction phase is complete, the prototype will then be subjected to the evaluation criteria outlined in 4.4.

Section 6.1 examines the technology used to create the prototype. An evaluation methodology is then discussed in Section 6.3, which describes the actual implementation stage of the prototype using the technologies from 6.1. Section 6.5 applies the technical criteria constructed in 4.2 to the completed prototype. Section 6.4 then applies the use case walkthroughs that were similarly constructed in the Evaluation Criteria section 4.1. Finally, Section 6.6 reviews this first prototype against the workspace transference framework and discusses the conclusions reached from this initial prototype implementation.

6.1 Technology selection

The first priority of the prototype implementation stage is to select a viable programming environment to use for the prototype implementation. In order to correctly select a suitable programming environment, the workspace transference framework must be analysed to generate an objective list of criteria that a candidate environment must satisfy.

The following list details how each framework requirement applies to each candidate environment and is drawn from the original framework requirements in Chapter 3:

- **Application portability:** Cross-platform application development of a compatible stand-alone executable (i.e. no prerequisite interpreter or launcher would be required to execute the prototype).
- **Data portability:** The ability to access data in a cross-platform–compatible way.
- **Data sharing:** The ability to access data remotely via networking in a cross-platform–compatible way.
- **Environmental interaction:** OS-level interaction in a cross-platform–compatible way.
- **UI requirements**: Direct or indirect (via additional libraries) support of CLI and GUI development.

Rank	Environment	Domain	Application portability	Data portability	Data sharing	Environmental interaction	Interface requirements	Candidate
1	C	Client	✔	✔	✔	✔	✔	✔
2	Java	Client, Server	✘	✔	✔	✘	✔	✘
3	C++	Client	✔	✔	✔	✔	✔	✔
4	PHP	Server	✘	✔	✔	✔	✘	✘
5	JavaScript	Browser, Server	✘	✘	✘	✘	✔	✘
6	Python	Client, Server	✔	✔	✔	✔	✔	✔
7	C#	Client, Server	✘	✘	✘	✔	✘	✘
8	Perl	Client, Server	✔	✔	✔	✔	✔	✔
9	SQL	Query	✘	✔	✘	✘	✘	✘
10	Ruby	Client, Server	✔	✔	✔	✘	✔	✘

Table 5. Comparison of programming environments against the workspace transference framework requirements

Table 5 shows how the top ten programming environments[106] apply to the workspace transference framework requirements. In each case, the environment has been assessed for its conformity to the above criteria and awarded either a pass or fail mark for that framework criterion.

From Table 5, the following conclusions can be drawn:

- **Domain:** In some cases, the environment's domain automatically exclude it from the selection: PHP, JavaScript and structured query language (SQL) are three such environments that are aimed at server-side development, browser functionality and DB queries, respectively. Since these domains are outside of the required client-side functionality, they must be deemed unsuitable and excluded from the selection process.
- **Application portability:** Five of the ten environments can compile a stand-alone executable. Java and C# are excluded from this selection criterion since both environments need the language framework to be installed on the host system in order to function.
- **Data portability:** Provided by a large selection of the candidate environments where the environment is not limited by the cross-platform incompatibility of the application portability requirement. The environments satisfying this criterion mirror those that passed the previous application portability requirement.
- **Data sharing:** Displays much the same level of compatibility as data portability with the exception of the SQL query language, which provides no networking functionality
- **Environmental interaction:** Provided by most candidates with the exception of Java (which is sandboxed), JavaScript (again sandboxed for the browser), SQL (a DB query language and thus no system access) and Ruby (very few host system interaction libraries available).

- **UI requirements:** Indicates that the majority of the top environments do in fact provide some method of a cross-platform UI. Notable exceptions to this are PHP (mainly a server-based environment), C# (Microsoft Windows-only UI libraries) and SQL (query language with no UI capability).

From this analysis, we can conclude that four environments are viable for the client-side prototype implementation stage: C, C++, Python and Perl.

An additional consideration within the prototype implementation is the issue of server-side integration stipulated in implementation of a server-side API (see 5.2.2). Of the four selected environments, only Python and Perl provide a workable framework for efficient server-side scripting. C and C++ do provide some integration in the form of CGI-based systems, but the actual implementation of such solutions is largely limited to web servers requiring a heavy degree of low-level optimisation. Python (via Django[107]) and Perl (via Mason[108]) both provide a client-side development environment alongside a rapid development server-side framework that is suitable for prototyping.

From the remaining two options of Perl and Python, Perl is supported by the Comprehensive Perl Archive Network[109] (CPAN) and provides a large selection of suitable software components that can be used to implement the modules outlined in the design stage. Python does offer a similar distribution network in its own PyPI project[110], but at the time of writing this repository, although it is promising, it lacks the selection offered by the Perl CPAN system.

Due to these considerations, Perl was selected as the first prototype development language due to its compliance with the workspace transference framework, its compatibility within the client and server domain areas and its choice of software to satisfy the modules proposed in the prototype design. Python is a close second in compatibility and can be considered a replacement should any issues be found with the Perl implementation.

A possible threat to validity exists in the selection of implementation environment. While Perl was used to create the initial prototype, it is possible that another environment discussed in Table 5 could also be viable for this research or could possibly produce different results. Perl and Python were selected as possible candidates for the implementation stage based on the above criteria with the understanding that a choice had to be made and it would appear that these two environments were the most viable.

6.2 Evaluation methodology identification

This section will discuss an existing evaluation methodology as a template during the prototype implementation stage.

Jedlitschka and Pfahl[111] proposed a possible prototyping research framework that was later revised by Kitchenham et al.[112]. This structure is critiqued in the seminal case study paper *Guidelines for conducting and reporting case study research in software engineering*[113], which provides the progression seen in Table 6 to analyse a prototype-driven academic work.

#	Sections	Chapter
1	**Title**	Title pages
2	**Authorship**	Title pages
3	**Abstract**	Chapter 3
4	**Introduction**	Chapter 1
5	**Related work**	Chapter 2
6	**Case study design**	Chapter 4
7	**Results**	6.6 & 7.4
8	**Conclusions and future work**	Chapter 8
9	**Acknowledgements**	Title pages
10	**References**	Chapter 2
11	**Appendices**	Chapter 1

Table 6. Reporting structure outlined in Runeson & Höst with the corresponding section references

Table 6 shows the case study layout proposed in the Runeson and Höst paper listed on the left side with the right side showing the corresponding chapter within this research paper.

While the majority of sections shown in Table 6 are common to research papers (specifically points 1–3 and 9–11), the major difference in prototype-driven research is in the execution of the case study design and evaluation methodology (points 6 and 7). These two major sections include generation of the testing methodology used in the construction of a case study along with its subsequent evaluation. Within the context of this research, the case study design (Section 6) is represented by the generation of evaluation criteria, the results of which (Section 7) are examined in the prototype implementation and its subsequent evaluation against the criteria specified in the case study design.

The Runeson and Höst paper also specified a checklist of items common to case study-based research. This case study checklist is itemised within the paper as a detailed list in Appendix A as well as a condensed version in Appendix B. Using the more general checklist specified in Appendix B as a baseline, the following points correspond to items within this research:

- Specification of objective and research questions addressed during the introduction (see 1.2), paper discussion and problem statement (1.1)
- Motivation for the solution, here the generation of a framework to address the existing deficiencies in workspace portability, is addressed with the corresponding workspace transference framework in Chapter 3
- Data collection procedures accomplished with persona creation and evaluation use case tests and technical criteria generated in the evaluation criteria (Chapter 4) and later against the constructed prototypes in Chapters Chapter 6 and Chapter 7
- Triangulation applied to both the persona use case tests (4.1) and the technical criteria (4.2) generated in the evaluation criteria
- No ethical issues present during this research
- Conclusions listed within Chapter 8, which includes a discussion on future research in Section 8.2

The Runeson and Höst prototyping model forms a viable template methodology, aspects of which will be used in the evaluation of the prototype solutions developed in the following implementation stages.

6.3 Framework compliance

This section discusses the initial prototype's adherence to the workspace transference framework (see Chapter 3). What follows is a breakdown of the framework requirements and how they relate to the prototype implementation phase after the design discussed in Chapter 5.

6.3.1 Application portability

The two modules required at this stage of prototype development (see 5.1.1), Package Support (5.2.1.1) and Emulation (5.2.1.4), were provided within the Perl development environment with its support of cross-platform networking (6.3.2) and use of IPC. IPC is needed for the prototype's communication with external processes such as the CygWin and WINE emulation environments as well as the later Unison sub-system used in the data portability synchronisation modules (see 6.3.2). While IPC is provided within Perl to a degree, neither Perl nor the later Python programming environments provides a consistent method of automating cross-platform IPC. Both Perl and Python use variants of the Expect POSIX-based system, which lacks support outside of UNIX-based systems, Microsoft Windows being one notable example. Expect's functionality is also fairly complex and unwieldy in cases where a less complex query/response structure is required, such as within the Unison application used in the prototype. Owing to these factors, a cut-down equivalent module called Predict was developed, which followed approximately the same principles as Expect but provided support for the target OSs.

In order to address the deficiencies in cross-platform IPC libraries, the Predict module was coded specifically to communicate with these executable sub-systems without having to rely on platform-specific technologies such as Expect. Predict was coded natively in Perl using the IPC::Run[114] library and in the later Python implementation (see Chapter 7) using an extended version of the built-in sub-process object system (Perl uses '::' to signify hierarchy so 'Run' is a child of the 'IPC project'). This IPC system allows basic process management within Python but lacks the advanced facilities of the POSIX-based Expect utility. In both cases, Predict is initialised with a list of rules on how to act when a given output occurs. These rules typically take the form of a matching pattern of either a simple string or a more complex regular expression. This matching condition would be followed by an action that represents a reply input back to the IPC sub-process. Using this methodology, any console-driven program can be encased in the Predict shell and run as if a user were operating the input and output. Predict is also fully thread-safe and allows parallel executions, features that are not provided by the Expect system.

Although it is unfortunate that Perl does not already have a cross-platform out-of-the-box Expect equivalent, the construction of Predict enabled the prototype to provide cross-platform IPC control for both the CygWin emulator for the running of Linux applications within a Windows environment as well as the WINE emulator for the opposite Linux OS.

As the original application portability requirement states, the prototype and the applications it manages should be relatively portable. This requirement was satisfied in the initial implementation by the cross-platform–compatible prototype core (in this case, a Perl pre-compiled object) and use of the platform's native Perl compiler. While not comprehensive, this method provides a relatively stable way to run the prototype on multiple platforms without extensive alterations. Further details on the prototype's portability and the methods used to allow native code execution on each host platform can be found in 6.5.1.

6.3.2 Data portability & sharing

As stated in the framework integration criteria for this section (see 5.1.2), minimisation of the communication overhead of each of the prototype instances is desirable. To achieve this, a differential algorithm will be employed to calculate the changes applied to each file contained within the prototype's portable environment. The most obvious candidate for such a task would be the seminal RDiff (RSync Difference)[69] algorithm, which forms the central part of the cross-platform RSync project and in which only computed differences need to be transferred between the client and server. Over time, the RSync library has been improved and optimised in its native form (it is now on its 32nd protocol release), leading to the construction of the LibRSync library common to most POSIX systems. The compiled RSync library allows for the computation of optimised differences on any POSIX-compatible OS.

Reliance on an external Unison utility eliminated much of the complexity in integrating the RSync libraries during the prototype implementation. Use of the external Unison project allowed for outsourcing of the synchronisation functionality (see 5.2.1.5) to an upstream project that has a well-established code base and good cross-platform support. As with the IPC used in the application portability modules (see 6.3.1), the prototype's synchronisation functionality was integrated using a combination of the Unison project and the Predict module (see 6.5.2). The use of an external project eliminated the need for a differential computation system to be developed from the ground up and instead allowed the prototype to use the strengths of the existing Unison project.

The prototype's data transfer requirements (see 5.2.1.7) were accomplished using Perl's relatively large collection of Internet access libraries. These range from the seminal LWP[115] to more specialised components such as the WWW::Mechanize[116] module, which provides a fully featured programmable browser complete with user interaction and cookie support. This allows for interaction with any web-based API to function much more smoothly by automating the stateful transfer of information required by most online API standards.

LWP by itself is the most widely used component and fully deserving of the reputation it holds as the most versatile and multi-faceted CPAN library. Most of the smaller libraries, such as the simplifier LWP::Simple[117], extend the core LWP object's functionality to provide easier interaction for smaller projects or simpler tasks for which use of the full system would be overkill. Additionally, LWP provides the most examples, detailed documentation and a substantial section in most major books on Perl web programming[118]. This documentation assisted in the efficient and easy implementation within the prototype.

6.3.3 Environmental interaction

Perl provides extensive support across multiple platforms and presents a strong modular design for cross-platform support. Some functionality was lacking that needed to be coded specifically by the developer, such as the Predict module used in the synchroniser module (see 6.3.1), but for the most part Perl was a well-supported language that usually provided everything needed for the prototype's cross-platform requirements.

Minor omitted functionality within Perl includes cross-platform support to monitor file updates in a suitably cross-platform manner. The Linux and Macintosh platforms use the iNotify[119] kernel-level sub-system, whereas Windows uses its own equivalent API[120]. Switching between these two file change monitoring systems was a minor consideration in the construction of the prototype in which a multiplexer was required to allow for standardised access to their features regardless of host system. The lack of an existing Perl module to perform this functionality was disappointing, but the actual integration of said module was a relatively simple task during prototype development.

With the two exceptions listed above (Predict/Expect for synchronisation and iNotify for file system monitoring), no further modules or considerations had to be applied in order to make the prototype compatible across multiple OSs. The Perl CPAN module system[109] is exceptionally good at providing modules that work on all OSs due to its enforced testing standards that enabled efficient creation of the prototype on multiple platforms.

6.3.4 UI requirements

The Wx system was selected during the prototype implementation stage as the more viable of the available GUI systems due to its provision of the required cross-platform support and the requisite flexibility across a wide variety of environments. Unfortunately this flexibility comes with the price of increased implementation time. Older GUI toolkit systems such as the Windows Forms-based Visual Basic programming language along with early versions of Java AWT use a simplified display system based on fixed coordinates to position various screen widgets. For example, a button placed at a set of coordinates never changes unless the application programmer explicitly instructs the widget to move. This is a simplistic method, but is easier to design and distribute as the application programmer positions the screen widgets at the designated positions in the knowledge that the application appearance not differ on any compatible platform.

Wx uses a flow-based system in which the programmer *suggests* the appearance of the GUI and the Wx libraries are responsible for actual layout, which changes according to the local display settings. Wx is much more adaptable to changes in font sizes, window sizes, window placement and actual screen resolution, the latter of which is an important factor with the smaller-screened netbooks. While the development of a GUI within the Wx system is more complicated, its actual appearance is much more fluid and provides the requisite adaptability across its supported platforms. Owing to this extra complexity, there is a heavy price for this portability, which usually takes a large quantity of coding time to perfect.

In the initial Perl (and the later Python) prototype, the WxGlade[121] design system was used to construct the interface and generate the Wx[80] GUI-specific code. For each window, a separate GUI control module was constructed that would apply the underlying logic to how the interface would be populated with data and act on certain user events. The Wx interface was first created for the Perl implementation and remains essentially unchanged into the later revised prototype (see 7.1.4), WxGlade providing the functionality needed to quickly switch between supported languages.

Since Perl is not a GUI-led development environment by nature, some areas of this development process are weakly implemented within the Perl development environment. Specifically, some areas of WxPerl are poorly tested and can display unpredictable results on some OSs. Large areas of WxPerl lack documentation and some of its quirks become visible only during the testing process on each targeted platform.

One such example is the drawing of nested containers, a method of arranging GUI widgets within a window. When operating on the Linux environment, the WxPerl GUI system automatically redraws and recalculates the window and all child widget sizes when the window is first drawn to the screen. This behaviour is not replicated on Mac or Windows-based OSs and must be initiated on these systems within the application. Likewise, window background colours are left to a WxPerl-supplied default of dark gray in Windows, whereas the Linux and Macintosh machines both share the default system theme default background colour.

WxPerl is intended to function identically on all supported OSs but does not actually do so when deployed. As a result, the inclusion of a GUI within the Perl development environment was problematic due to the large quantity of testing required on each deployable platform. The prototype functioned as intended as evidenced in the persona walkthroughs (see 6.4), but the sheer number of bugs encountered during the testing phases places doubt on the long-term viability of development using the WxPerl GUI toolkit.

6.3.5 Framework compliance summary

Perl provides a strong cross-platform–compatible development environment but its complexity, code illegibility and small-scale problem area emphasis make its use unwieldy with larger projects. While all programming languages intend to provide an overall balanced approach to program design with the intention of providing a scalable implementation, some can quickly become strained by maintenance issues.

Perl also lacks a decent cross-platform compiler, making the distribution of stand-alone projects difficult. Its inefficiencies on non-UNIX systems show its obvious bias towards this platform, but after the initial interpreter bootstrap has been completed, Perl can run at speeds comparable with those of compiled C code.

6.4 Persona use cases

This section applies the use case walkthroughs generated in the evaluation criteria (see 4.1) against the prototype.

Except where otherwise stated, all screenshots are drawn from a clean installation of the Ubuntu Linux 9.10 OS but apply equally to all other operating environments due to the prototype's cross-platform compatibility. When necessary, Microsoft Windows XP will be substituted as the secondary demonstration OS (e.g. during cross-platform testing in 6.4.2.5).

Each task is broken down into activities conducted by that persona followed by any relevant screenshots illustrating progression though the task.

During this process, the workspace transference framework requirements (see Chapter 3) are referenced to show the prototype's satisfaction during the use case walkthroughs. The requirements listed in the framework are referenced by name and correspond to the descriptions given during the framework construction (see Chapter 3).

6.4.1 Novice Persona

The following walkthroughs evaluate the tasks associated with the Novice persona that were generated in 4.1.1. Since this persona represents the lowest technical ability level, the tasks conducted for this persona represent the most basic use of the prototype.

6.4.1.1 TASK: Install the prototype on a desktop machine

An easy and efficient prototype installation sequence is crucial for the distribution of the prototype to this user population. The below task progression demonstrates this procedure for the Novice persona but is equally applied for the Knowledge Worker (see 6.4.2.1) and Developer (see 6.4.3.1) personas.

1. The prototype is provided to the persona on a CD-ROM. Once the CD-ROM is placed in the computer's disk drive, the prototype installation process begins automatically.
2. The prototype's UI presents the initial installation sequence shown in Figure 6.
3. USB disks are automatically detected and added to the portable device list, and pressing of the refresh button beings a rescan of the attached disks if needed. Should another location be required (as with the Knowledge Worker persona below), it can be selected form this list.
4. A username and password combination are entered.
5. The persona clicks the 'OK' button or presses the Enter key.
6. The username and password are checked. If any inconsistency is found, a message is displayed.
7. If the username and password are valid for either a new or an existing account and the installation location is writable, then the installation begins as shown in Figure 7.

8. A simple message is displayed to the persona (Figure 8) as the prototype begins its full execution.

All of these task steps are intended to satisfy the UI requirements of the workspace transference framework (see 3.1.5) so the prototype installation can be easily accomplished by a user of any ability level. The installation sequence also conforms to the application portability requirements (see 3.1.1) by being transported in a single executable file that can be easily downloaded from the Internet for installation. This single file essentially represents a portable program in a single executable binary file that expands into the fully featured prototype environment.

A minor trade-off has been made for the initial installation sequence in that the host OS-specific installer should be selected. For instance, the Windows installer should be selected for Windows and Linux version for Linux. Unfortunately, this distribution method is unavoidable as there exists no suitable standard for simultaneous compilation of an application in a suitable way for all of the prototype-supported OSs. A possible future solution to this problem is the use of a CD-ROM to distribute the installation program. This disk can contain all three variants of the installer and select the appropriate executable for the OS (using various cross-platform Autorun rules).

After the prototype installation is complete, programs from all three OSs can be run interchangeably as required by the application portability requirement specified above.

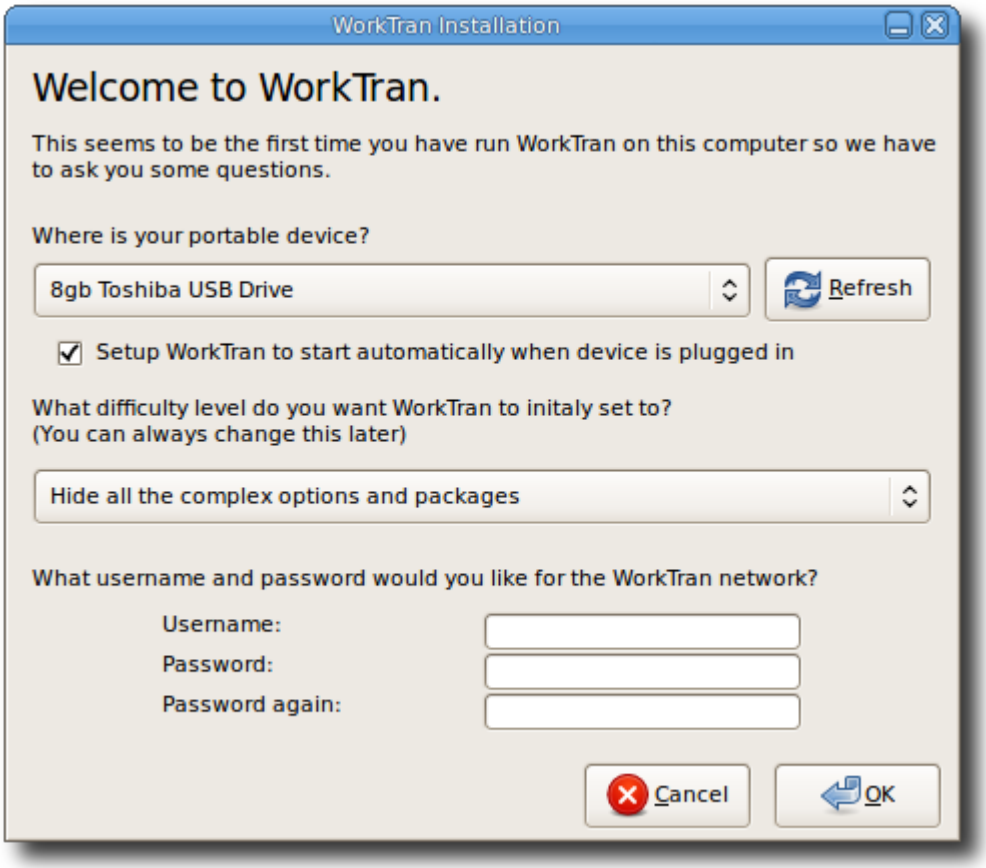


Figure 6. The prototype installation sequence presented in its initial form

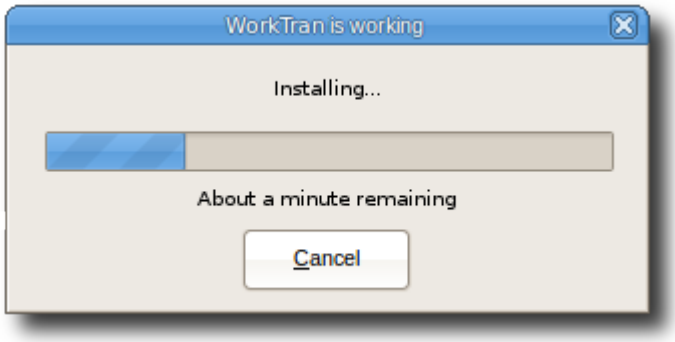


Figure 7. The prototype installation begins

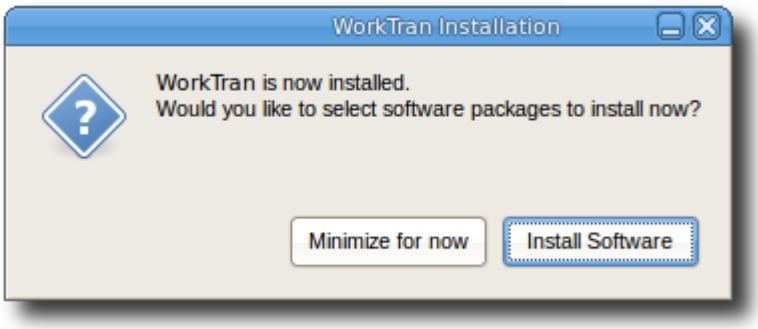


Figure 8. Prototype installation is completed

6.4.1.2 TASK: Install a free virus checker package

As with the installation sequence (see 6.4.1.1), the virus checker installation satisfies the application portability requirement (see 3.1.1) of the workspace transference framework. This process, illustrated below, allows for easy installation of portable applications with little difficulty regardless of the user's ability level. The method chosen below is relevant to the Novice and Knowledge Worker personas (see 6.4.2.2) in that it presents a friendly GUI interface. A less visually appealing but automatable version of this installation sequence can be found for the Developer persona in 6.4.3.6, which demonstrates a console-based alternative for the initial and package installation sequences.

1. This task can be accomplished in one of two ways: either the persona selects the shortcut to the package installer from the installation confirmation box in Figure 8 or they double-click the prototype system tray icon to launch the package installer.
2. The package installation screen is displayed (see Figure 9).
3. The persona can either select the virus scanner from the default list or select a category (as in Figure 9 where 'Popular' is selected) followed by the item.
4. When the persona clicks on the virus scanner, the status bar is updated ('AVG Free is not yet installed') along with an information pane displaying a general description of the package.
5. The persona can right-click on the item in the list or click the 'Install AVG Free' button to begin the package installation. The status is shown in much the same way as that of the initial installation (see Figure 10).

6. Since this particular package automatically functions without any configuration, the installation sequence finishes here with a minor message displayed to the persona indicating that it is installed. The status line changes to indicate that the software is installed in the same style as later displayed in Figure 16.

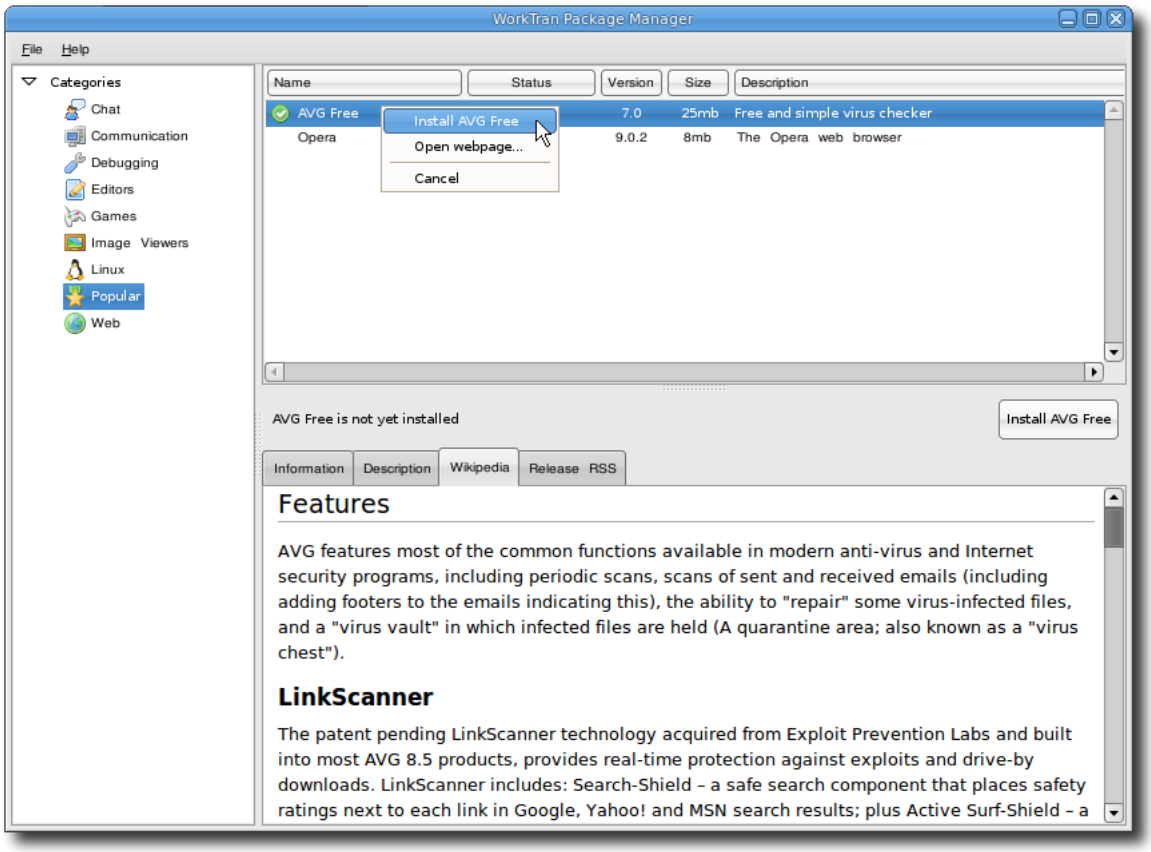


Figure 9. The prototype package installation screen

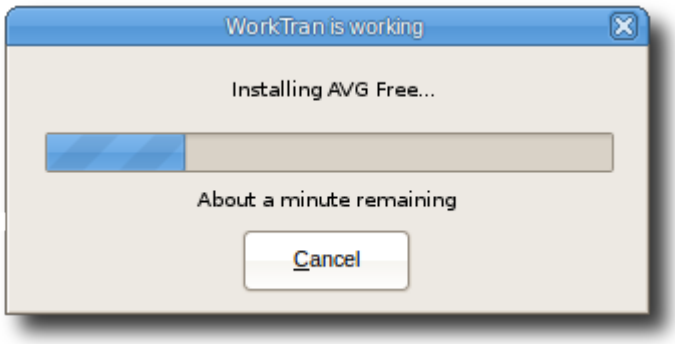


Figure 10. Installation status of a specific package

The package installation sequence demonstrates a clear advantage for users of the prototype over traditional installation. While platforms such as Linux already possess a suitable package management system, this is the first attempt at bringing this functionality to a Windows-based OS that is not biased to one particular brand (e.g. the *Adobe Updater* service, which provides package-like updates for Adobe products and *Windows update,* which performs the same for Microsoft products).

In addition to being the first of its kind available to the Windows platform, the package management system is further strengthened by its compatibility with applications from other OSs. Applications can be installed from any of the supported systems, such as Linux-based applications on the Windows platform or the reverse. Such functionality drastically increases the number of applications available within the prototype's environment and opens the route for interoperability between OSs within the prototype.

6.4.1.3 TASK: Install a word-processor package

The word processor package installation task operates in much the same way as the virus checker package installation. See 6.4.1.2 for further details.

The exception to this task is the final step where instead of display of a simple 'Installation Successful' message, the dialog box shown in Figure 11 is displayed to allow the optional execution of the newly installed application.

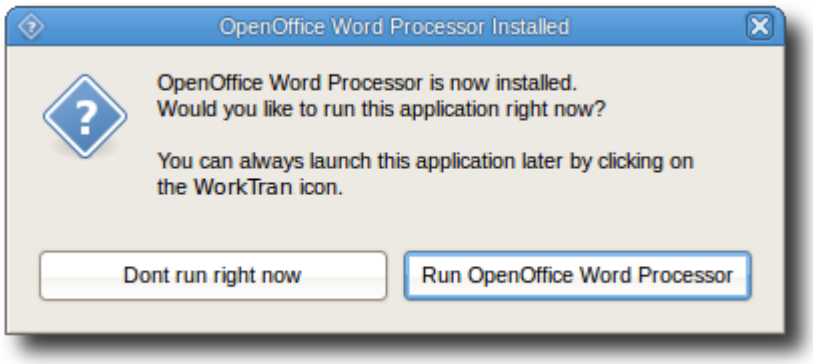


Figure 11. The word processor installation of is complete

6.4.1.4 TASK: Type and save a simple letter

The task of creating and manipulating a word processing package shows the functionality of the application portability requirement of the workspace transference framework. In the simple letter composition task, installation of the OpenOffice word processor (see 6.4.1.3) is seen to operate on all OSs, allowing the user to interact with the software in the same way and to the same end, saving or opening files from the prototype's environment.

1. The persona clicks on the prototype system tray icon.
2. The persona selects the 'OpenOffice Word Processor' from the open list as shown in Figure 12.
3. The word processor opens. Text is typed into the word processor window.
4. The resulting document shown in Figure 13 is saved to a file.

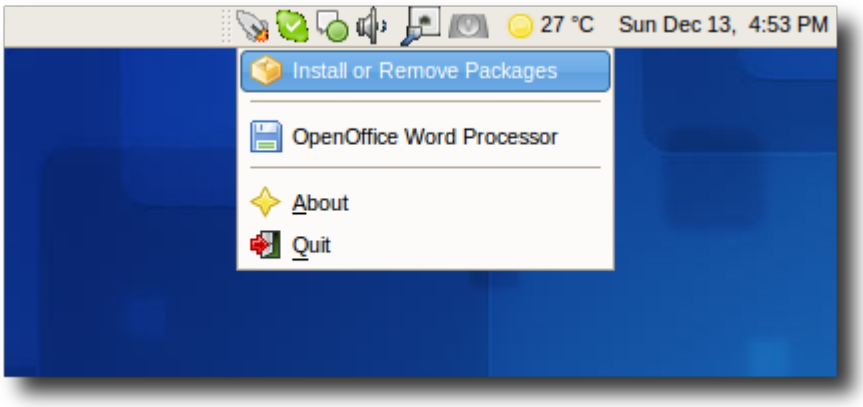


Figure 12. Selection of the system tray icon displays a list of installed software

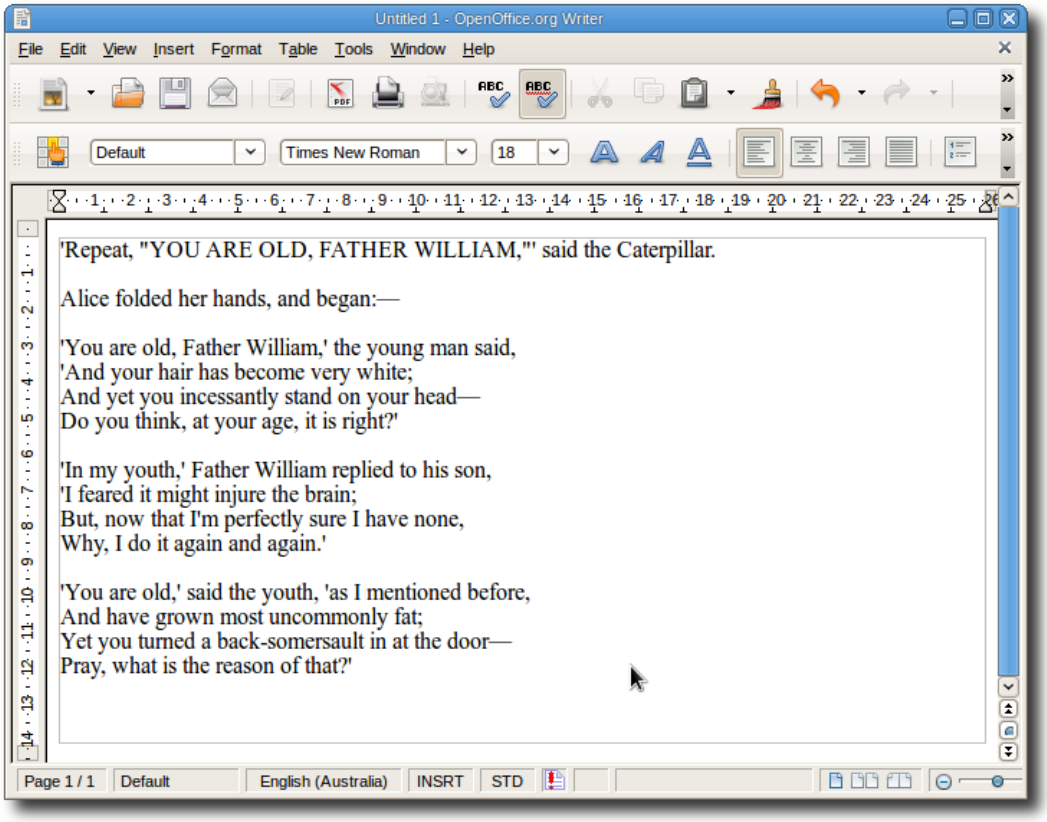


Figure 13. Unsaved work in the OpenOffice word processor

Use of the word processing application as outlined above is an example of standard user interaction with a desktop-based application. The prototype expands on this scenario in that any application run within the prototypes environment performs in the same way regardless of host OS environment or portability situation. If the prototype was now to be transported on a USB stick to another OS, the exact same functionality would be presented: the prototype would automatically emulate and provision OS-specific functionality without requiring any user intervention.

This functionality demonstrates the obvious application of the application portability, data portability (the files being moved along with the environment) and environmental interaction (the same behaviour on all OSs) aims of the prototype.

6.4.1.5 TASK: Change the document contents

This task demonstrates the recovery of data from a fairly common scenario, that is, the overwriting of *good* data with bad. While file recovery is sometimes possible for deleted files, recovery of overwritten data is rarely successful as the file system, in order to save disk space, often uses the same storage location on the disk when re-saving an open file. Overwriting data in the way demonstrated here usually means that the data has been permanently lost.

1. The document saved in 6.4.1.4 is re-opened either by re-launching the word processor or directly opening the saved file.
2. Text is overwritten in the document as shown in Figure 14.
3. The (now corrupted) file is re-saved by the persona back to the disk.

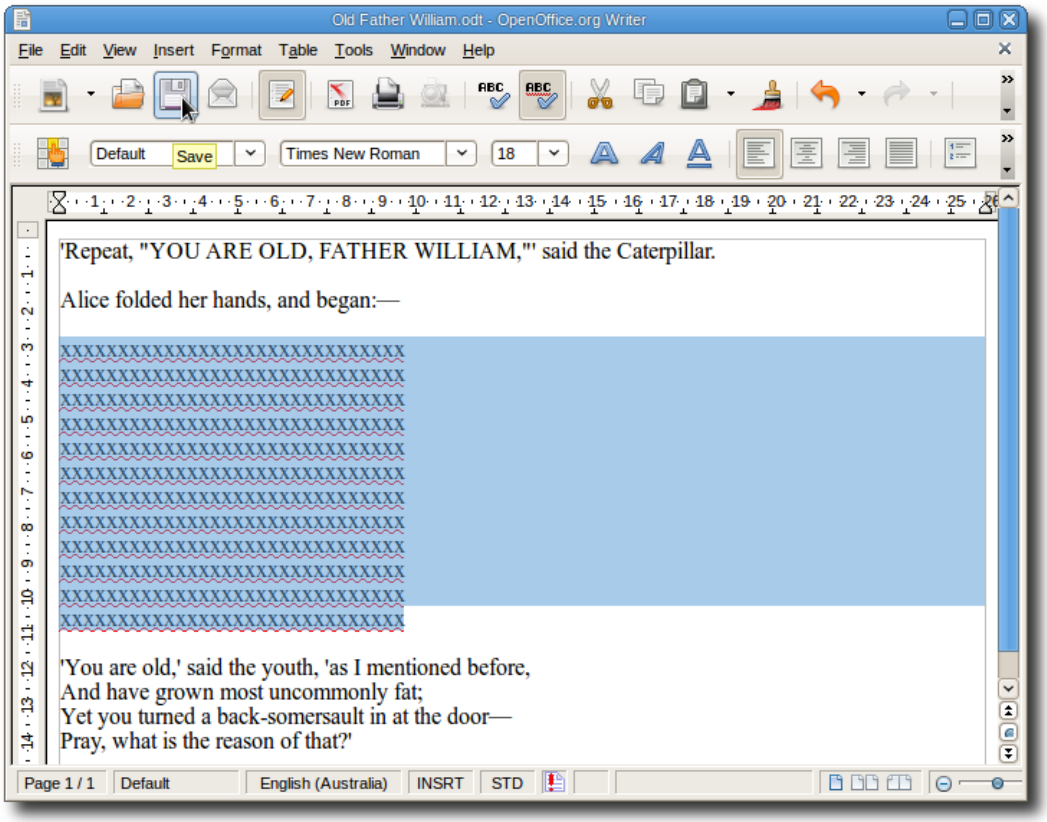


Figure 14. The same document now with purposely overwritten text

6.4.1.6 TASK: Attempt to recover the first version of the file

Continuing from the previous example of common data loss shown in 6.4.1.5, this task demonstrates that the prototype can be used to restore data from the scenarios outlined in 6.4.1.5 using its central server storage.

1. The persona opens the prototype server's website.
2. The file created in 6.4.1.4 is selected.
3. Various versions of the file are displayed.
4. The relevant and earliest file version (i.e. the correct fully populated file) is selected and the Restore action is selected.
5. The file is now restored to the previously un-corrupted state upon the next synchronisation.

Scenarios such as this can occur as further common mistakes like accidental file deletion, overwriting files as demonstrated in 6.4.1.5, disk failure and other data loss. In all of these cases, the prototype successfully demonstrates the data security workspace transference framework requirement (see 3.1.3) by providing a secure and suitable method to restore data from any point in its lifetime.

6.4.2 Knowledge Worker Persona

This use case walkthrough describes the tasks performed by the Knowledge Worker persona that were generated in 4.1.2.

This persona represents the intermediate computer ability level and assumes that this persona is capable of all the tasks outlined under the Novice persona (6.4.1) as well as the example scenarios outlined in this section.

6.4.2.1 TASK: Install the prototype on a netbook machine

Since the installation sequence is largely a replication of that conducted in the Novice persona (see 6.4.1.1), very little in the installation stage is different for the Knowledge Worker except for slight extension of some otherwise hidden options. In this scenario, which occurs during the installation of the prototype, the persona will install the prototype on a single already-portable machine such as a netbook or laptop rather than a USB stick. As can be seen from the below installation sequence, this has little extra complexity and again demonstrates the prototype's data portability.

1. As with the Novice persona, the installation sequence is executed by the persona (see 6.4.1.1).
2. The persona chooses the 'Install on this computer only' option rather than any detected USB drives. If no USB drives are present in the system, this option is selected by default.
3. The sub-selection for which folder should be managed becomes visible as in the above Figure 15, where the 'My Documents' folder is suggested.
4. Installation continues in much the same way as for the Novice persona (see 6.4.1.1).

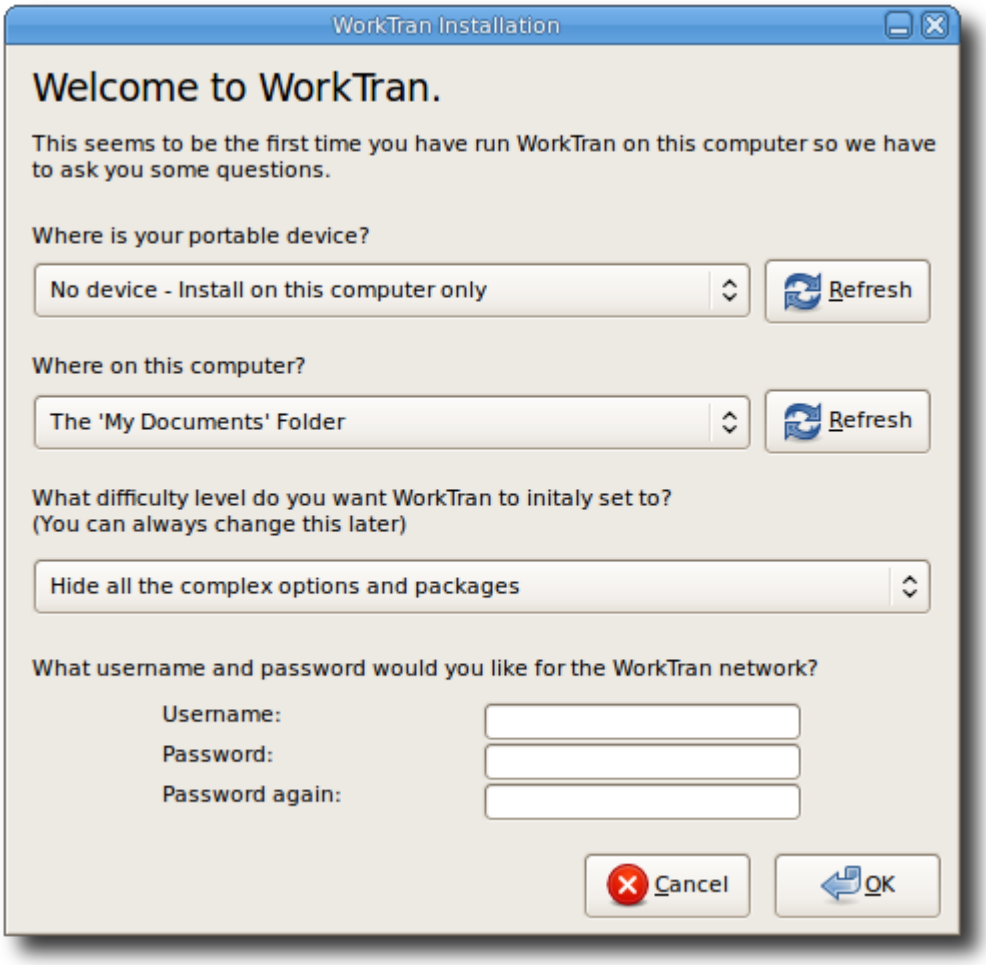


Figure 15. Extended installation screen showing more options

6.4.2.2 TASK: Install web browser package

As with the initial prototype installation sequence shown in 6.4.2.1, the web browser package installation task shows a similar process to that of the word processor installation sequence for the Novice persona (see 6.4.1.2).

1. The persona selects the 'Install or Remove Packages' option from the system tray as shown in Figure 12.
2. The persona selects the Firefox package and clicks the 'Install Firefox' button.
3. The package is installed and the package manager window updates as shown in Figure 16.

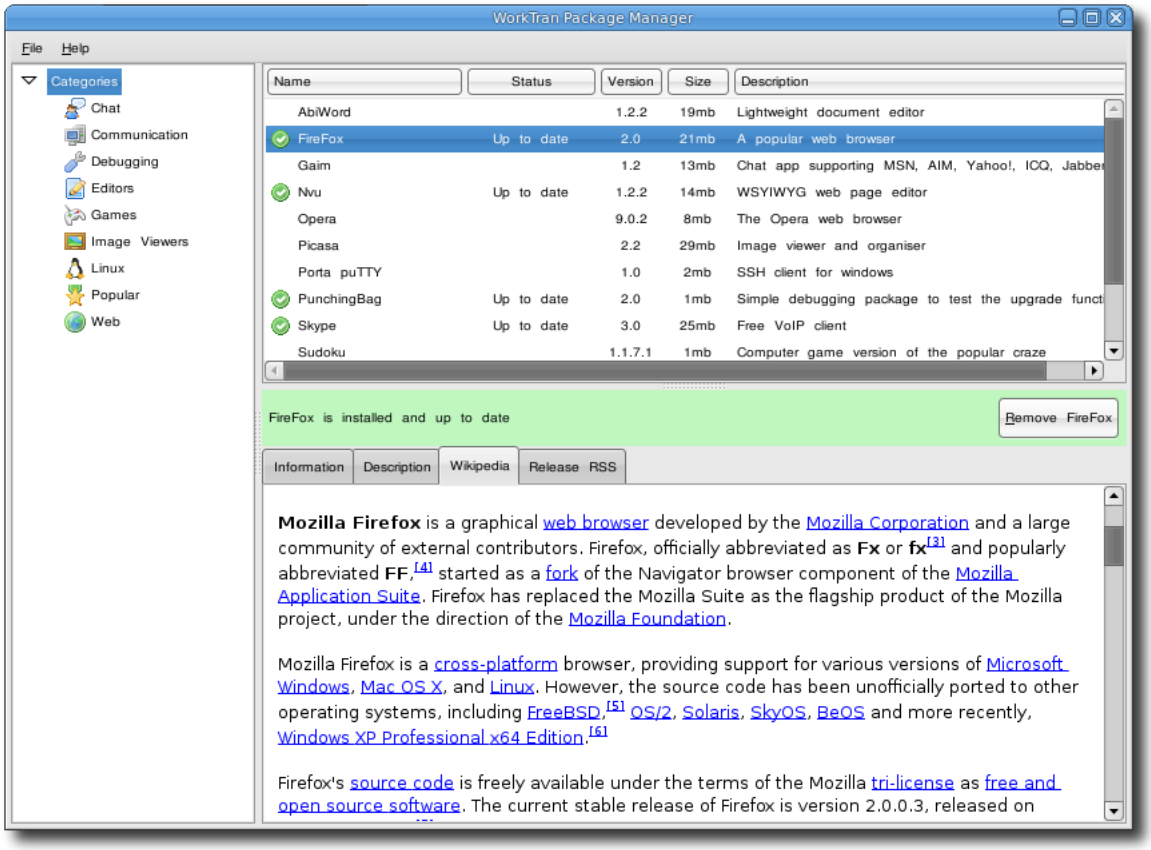


Figure 16. The now installed Firefox package is updated in the package manager window

6.4.2.3 TASK: Use web browser to bring up a web page

As with the use of a word processor application for the Novice persona (see 6.4.1.4), this web browser operation task demonstrates the support of a complex application across multiple platforms.

1. As with the OpenOffice word processor demonstrated in Figure 12, the Firefox application is launched from the system tray.
2. The application is launched (as shown in Figure 17) and usable as a portable application.
3. To test the cross-platform functionality relevant in the next step, the Adblock and TabMixPlus plug-ins are installed.
4. The persona visits the following websites using the web browser: nytimes.com, slashdot.org, google.com.au and digg.com and finally clicks the 'Home' button to return to the main home page.

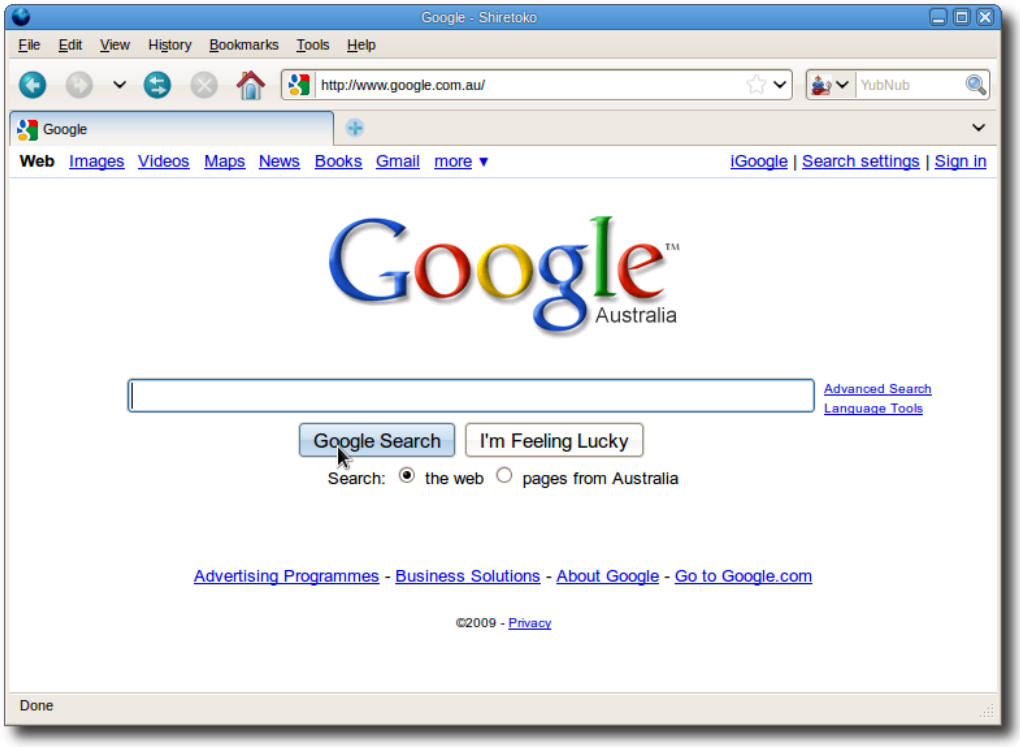


Figure 17. The installed Firefox package's main window

As with the previous example of a word processor (see 6.4.1.4), this task demonstrates the application portability requirement of the workspace transference framework. In addition to enabling the browser to run within multiple environments, use of the browser also demonstrates the environmental interaction framework requirement. Since the web browser can be run on multiple machines or within multiple environments (the host OS, for example, connects to different networks while being moved around), the prototype attempts to provision each of these different environments. Within a work environment, for example, a wired connection may be used in which a proxy server is used to provide web access. In this case, the prototype supplies the web browser (in this case, Firefox) with those relevant proxy details. At home, for example, a similar Internet connection may be used where no proxy is present and the Internet connection is provided directly by the user's ISP. In the latter case, the prototype again configures Firefox with the correct connection details.

This environmental interaction feature demonstrates that the prototype not only allows for an application to be moved between environments but also provides for the environment to itself be moved. In all of these cases, the prototype successfully provisions for each scenario and allows the applications contained within the prototype's environment to function correctly and with no additional effort from the user.

6.4.2.4 TASK: Install the prototype on a desktop machine

This installation of the prototype on a desktop machine is essentially a duplicate of the similar 6.4.2.1 task. In order to test the prototype's cross-platform capabilities, the desktop machine runs the Microsoft Windows XP OS. Since the prototype executes in much the same way, the installation sequence correspondingly also performs in a similar way and therefore requires the same input as the previous installation tasks within the Ubuntu Linux OS.

6.4.2.5 TASK: Open the web browser and view history

As with the initial web browser usage examples (see 6.4.2.3), the task of using the web browser in an entirely separate prototype installation demonstrates that the prototype can provide the application portability (for the different environment in which the prototype is now installed), environmental interaction (for the local system settings that the browser is configured with) and data sharing.

Data sharing is used here to demonstrate that two versions of the prototype (the first as shown in 6.4.2.1) were installed on the user's netbook computer, while this one is installed on a desktop machine. On both these machines, the actions performed on one are exactly mirrored on the other. This allows the user to interact with whatever machine is required at the time and the prototype correctly works around the user's behaviour.

1. The web browser is launched on the desktop machine.
2. As seen in Figure 18, the web browser has traversed the system boundary and now functions along with the Adblock plug-in (a small red stop sign in the top right corner) and the TabMixPlus plug-in (appearance of the tab bar).
3. The history from the previous session on the netbook machine, having been synchronised between the two workplaces, can now be seen.

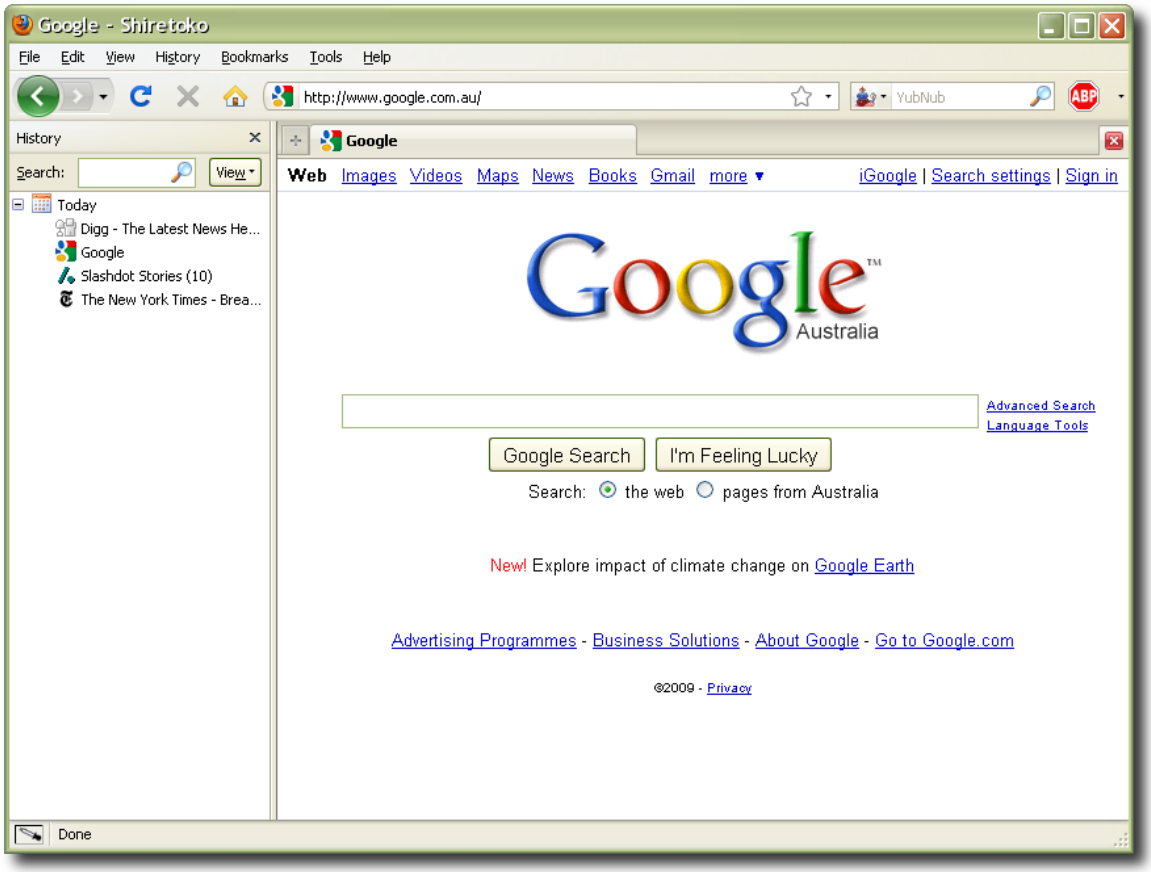


Figure 18. The same web-browsing session now installed on a Microsoft Windows machine

The data sharing requirement is demonstrated in this task using two machines, a netbook (originally set up in 6.4.2.1) and a desktop (set up in 6.4.2.4). While this demonstration only shows these two machines featured within the user's workplace, any actual number of machines can be brought into such an arrangement. Additional machines may include those of the workplace or home. The prototype manages them all and provides a standardised environment as if each machine were physically the same computer.

The ability to move between individual workspaces regardless of OS or local restrictions is not yet available in any other technology or research and clearly shows the unique provisions of the prototype and the usefulness of the workspace transference framework used to construct it.

6.4.3 Developer Persona

This uses case walkthrough to describe the tasks performed by the Developer persona that were generated in 4.1.3.

As with the previous Knowledge Worker example (6.4.2), this section assumes that the Developer persona is capable of the tasks conducted by the intermediate-level Knowledge Worker and the lower-level Novice personas (6.4.1).

6.4.3.1 TASK: Install prototype on a desktop system

Installation of the prototype on a desktop system task is a duplicate of the similarly named task within the Knowledge Worker use case (see 6.4.2.4).

6.4.3.2 TASK: Install GVim and the Perl scripting language

As with 6.4.3.1, GVim and Perl package installation is completed in much the same way as shown in the above Novice (see 6.4.1.2) and Knowledge Worker (see 6.4.2.2) package installation use cases.

Like the 6.4.3.1 installation sequence for the Developer persona, this task exists to set up the prototype within a regular desktop environment to enable later Developer persona tasks.

6.4.3.3 TASK: Disconnect from the Internet

The Developer persona disconnects from the Internet with no complaint or issue being raised by the still-executing prototype. This is a relatively basic task that serves to demonstrate the offline compatibility requirement of the workspace transference framework. Conversely, the continued execution of the prototype without any issues demonstrates that this aim has been accomplished.

6.4.3.4 TASK: Compile a simple script on Linux

The program compilation task attempts to show the more complex areas of the prototype's functionality. The satisfaction of the application portability requirement is shown from the perspective of a programmer creating a new application rather than a pre-packaged application shown in the above demonstrations.

A simple demonstration program is created from the following Perl script:

```
#!/usr/bin/perl
print "$_ bottles of beer on the wall...\n" foreach reverse 1..3;
print "No more beer\n";
```

This is a relatively simple script designed to display an output followed by termination. For space reasons, the script is kept intentionally simple and simply outputs the following to the console when run:

```
3 bottles of beer on the wall...
2 bottles of beer on the wall...
1 bottles of beer on the wall...
No more beer
```

This script is intended to provide a simple measurable output when moved between OSs in the later 6.4.3.5 task.

After the above script has been saved, the Internet connection is re-established and the files are synchronised to the server.

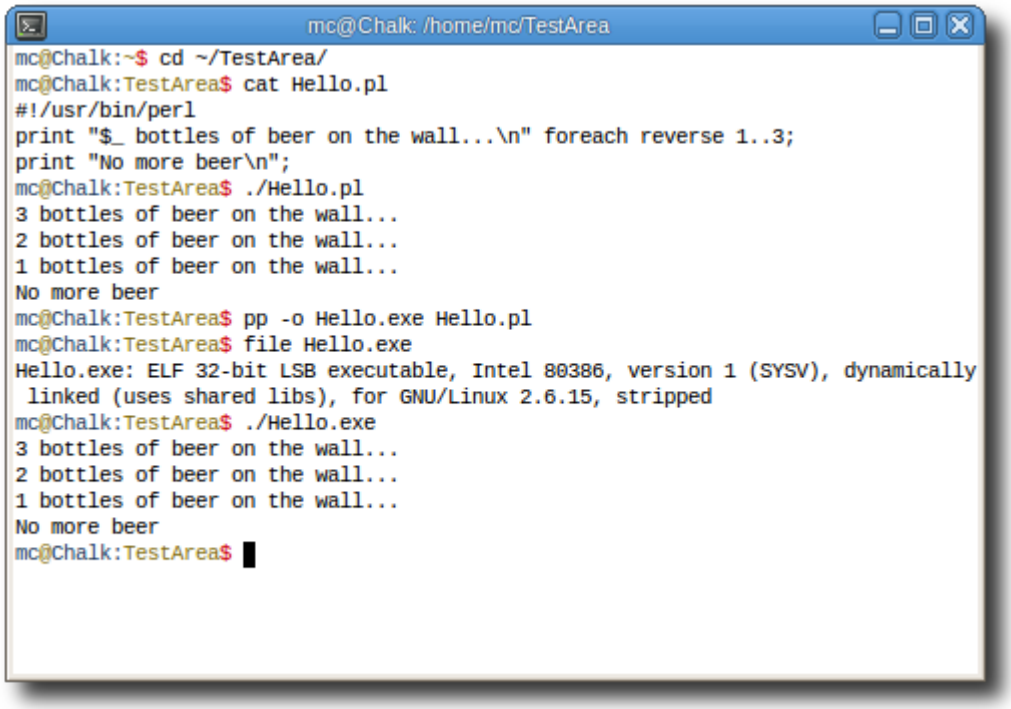


Figure 19. The result of a simple Perl program being compiled on an Ubuntu Linux system

As seen in Figure 19, this program is compiled into a Linux standard ELF format that is compiled and executable on that platform.

6.4.3.5 TASK: Demonstrate the application on Windows

After the initial creation of a binary program in 6.4.3.4, it is now possible to demonstrate the prototype's functionality by executing the same Linux ELF program on the Windows platform:

1. Installation of the prototype onto the Windows platform has been omitted here for reasons of brevity but follows the same process as listed in 6.4.2.4.
2. As shown in Figure 20, the same program runs successfully on the Windows platform using the portability handlers that intercept the executable and pass the program though the CygWin emulator to interpret the Linux-specific code on the Windows platform.

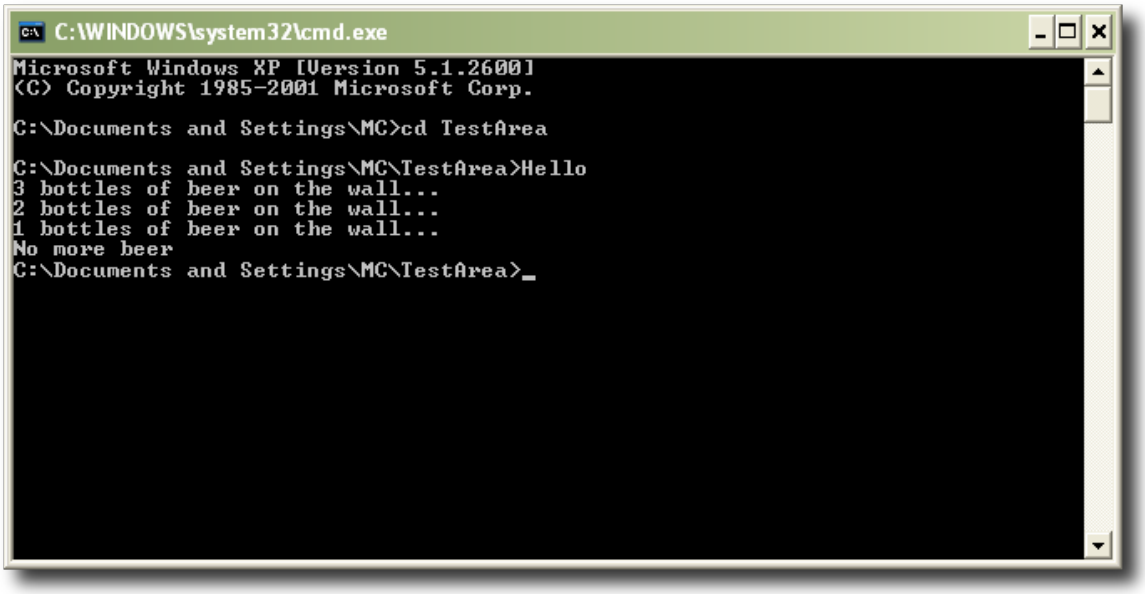


Figure 20. The same application running under the prototype's portability environment

The success of this task might be understated by the simplicity of the test application, but the prototype is performing various emulation and translation functionality internally to enable such cross-platform interaction. The effortless transference of a program from one machine to another is one of the successes of the prototype and is demonstrated here in that both a packaged application and user-created content can be transported.

6.4.3.6 TASK: Install prototype on a remote Linux server

In addition to the regular GUI installation sequence, the CLI interface created during the implementation phase (see Chapter 6) can be used to install the prototype within machines in which a full GUI environment is not available:

- Installation on a remote server can be accomplished in two ways. The easiest method is to use an X forwarding device, which allows the prototype to install onto the remote server but presents the GUI interface on the local machine. This interface is the same as that presented in 6.4.1.1 for the Novice persona. Alternatively, a console-based approach can be used if no GUI rendering system is present. This method is used to demonstrate the ability of the prototype to work in non-graphical environments.
- As shown in Figure 21, the prototype can be installed within a console-driven environment through simple running of the prototype's executable. The prototype detects the inability to use a GUI system and reverts to a console-driven installation instead.

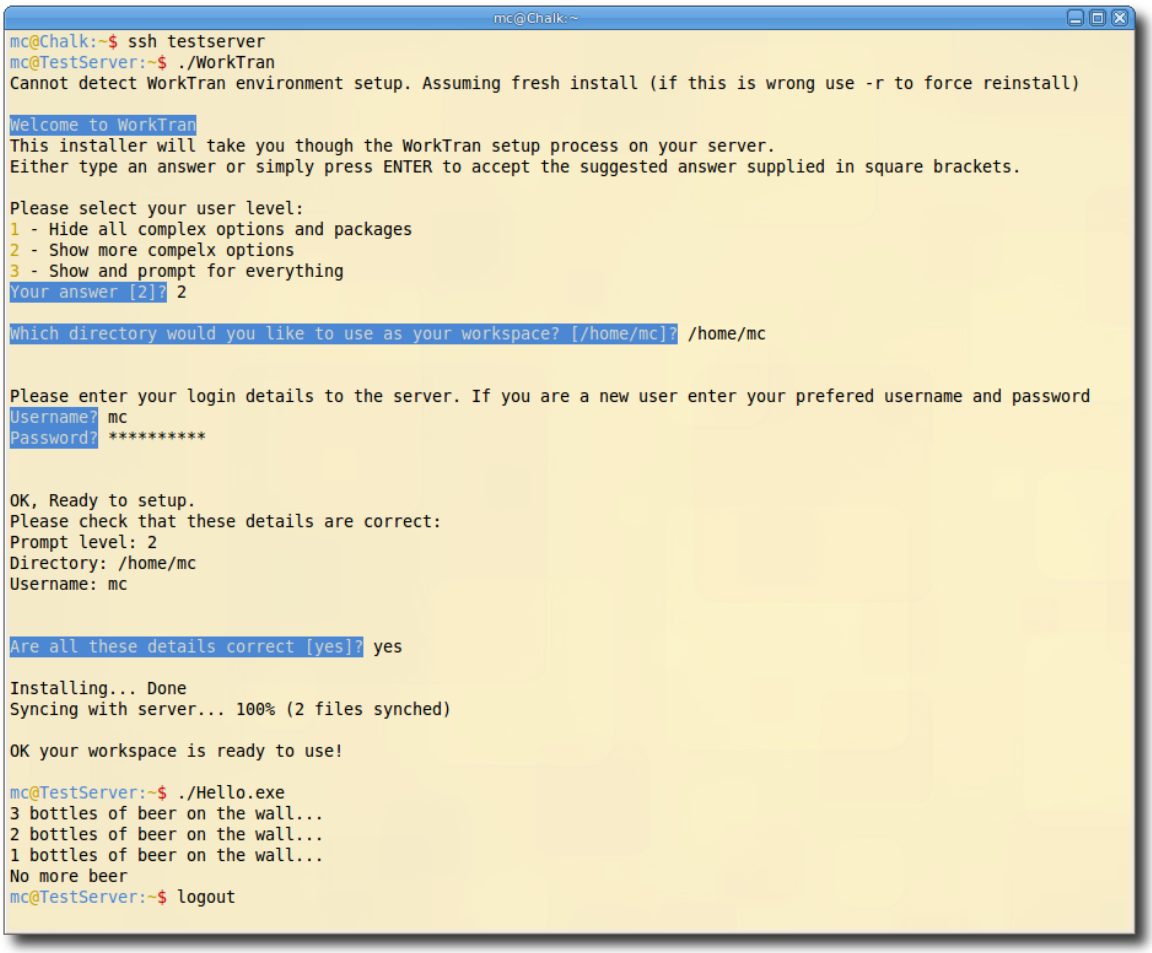


Figure 21. Installing the prototype within a purely console-driven environment

As demonstrated above, the extra provision of creating a CLI-based environment allows the prototype to be used within less graphically able machines. While not indicative of its ease of use, this outcome nonetheless shows that potentially any machine can be supported despite its inherent restrictions.

6.4.3.7 TASK: Create a package from the application

The package creation task shows that the files created in the previous task (Hello.pl and Hello.exe) can now be turned into a package and uploaded to the package management system.

The same process was used to package all of the available content within the package manager available in the earlier installation sequences.

As the process shown in Figure 22 shows, the created files can be easily turned into a distributable package and then uploaded to the server for installation within other environments.

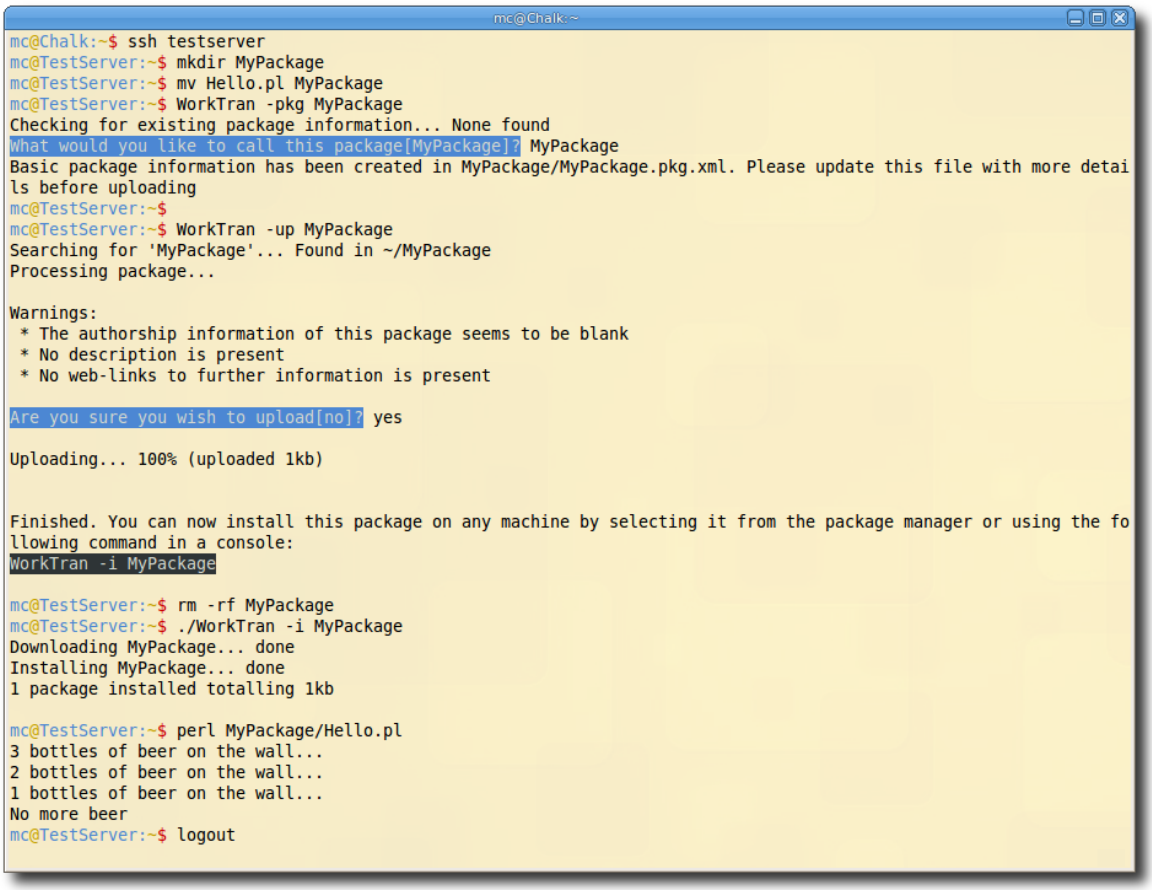


Figure 22. The package creation process as seen in the console

6.4.4 Persona use case summary

The use case walkthroughs shown for the Novice, Knowledge Worker and Developer personas demonstrate the practicality of the prototype for these three groups. Likewise, the practicality of the workspace transference framework is shown in that it can be used to successfully construct a working prototype to show the real-world practicalities of the functioning application.

As discussed in 4.1.4, there are a number of possible validity threats in the form of rejected personas not included as persona tests in 6.4. The inclusion of rejected persona groups and expansion of the suggested task lists for the three existing personas in a future research project would comprehensively test the initial prototype. Different or more varied tests could possibly generate other results than the output shown on more varied technical platforms under more complex situations. Another possible avenue of research is to expand the persona-based testing into user-based testing. This would require various ethical clearances as well as a comprehensive review mechanism to process the results. As with the varying personas and tasks, user-based testing could also produce different results from those discovered in 6.4.

The three selected personas from 4.1 all performed the allocated tasks according to the original testing plan created in Chapter 4. The following list demonstrates the various workspace transference framework requirements and how they were satisfied by the relevant use case tasks during the persona-based testing:

- **Application portability:** Application portability across OSs is shown from the initial installation sequence (see 6.4.1.1 and 6.4.2.1) where the prototype is provided as a single executable file to set up the full workplace environment. Packaged applications are shown in their cross-platform formats with installation of the AVG Virus Checker (6.4.1.2), the Firefox web browser (6.4.2.2) and the OpenOffice word processor (6.4.1.3).
- **Data portability:** Moving data and applications between hosts can be seen in the transportation of the prototype between portable media (see 6.4.1.1), netbooks (see 6.4.2.1), desktops (see 6.4.2.4) and remote servers (see 6.4.3.6).
- **Data sharing:** Data sharing can be seen in tasks 6.4.3.4 and 6.4.3.5 where a programming project is demonstrated to be simultaneously conducted on two separate machines.
- **Environmental settings:** The importing of system settings can be seen in tasks 6.4.2.3 and 6.4.2.5 where the Firefox web browser successfully imports system settings such as network configuration details. System interaction such as the GUI interaction with the host system can be seen in all of the above application portability requirements. File system interaction is demonstrated in the programming tasks 6.4.3.4 and 6.4.3.5.
- **UI requirements:** Offline compatibility can be seen in task 6.4.3.3. All of the above tasks successfully show consistent ease of use. Specific detailed references for lower ability level users can be found in the Novice persona walkthroughs, including the initial installation (see 6.4.1.1) and regular use (see 6.4.1.4).

The above framework failings can be analysed further using a table to illustrate the issues with each persona test.

Chapter	Persona Use Case	Issue	Issue Section
6.4.1	**Novice persona**		
6.4.1.1	Install the prototype on a desktop machine	Bootstrap	7.2.1
6.4.1.2	Install a free virus checker package	Predict	7.2.2
6.4.1.3	Install a word processor package	Predict	7.2.2
6.4.1.4	Type and save a simple letter		
6.4.1.5	Change the document contents	Predict	7.2.2
6.4.1.6	Attempt to recover the first version of the file	Web interface	7.2.3
6.4.2	**Knowledge Worker**		
6.4.2.1	Install the prototype on a netbook machine	Bootstrap	7.2.1
6.4.2.2	Install a web browser package	Predict	7.2.2
6.4.2.3	Use web browser to bring up a web page		
6.4.2.4	Install the prototype on a desktop machine	Bootstrap	7.2.1
6.4.2.5	Open the web browser and view history	Predict	7.2.2
6.4.3	**Developer**		
6.4.3.1	Install prototype on a desktop system	Bootstrap	7.2.1
6.4.3.2	Install GVim and the Perl scripting language	Predict	7.2.2
6.4.3.3	Disconnect from the Internet	Exceptions	7.2.4
6.4.3.4	Compile a simple script on Linux		
6.4.3.5	Demonstrate the application on Windows	Predict	7.2.2
6.4.3.6	Install the prototype on a remote Linux server	Bootstrap	7.2.1
6.4.3.7	Create a package from the application		

Table 7. Issues uncovered during the initial prototype implementation

Table 7 shows the list of persona tasks and the issue with each. Some of these issues such as the bootstrap time (present in all install operations) and the uses of PyPredict (used in all synchronisation tasks) are duplicated for similarly related tasks. Others such as the use of Django for server-based interaction and the use of Python's exception handling only occur in one persona-specific task. Some of these issues such as the bootstrap time (present in all install operations) and the uses of PyPredict (used in all synchronisation tasks) are duplicated for similarly related tasks. Others such as the use of a web interface for server-based interaction and the use of Python's exception handling only occur in one persona-specific task.

The issues discovered during the persona testing phase included:

- **Bootstrapping:** The slow start-up time of the prototype on Windows platforms represents a detrimental experience during the persona testing stages. The exact nature of this issue is discussed further in the distribution technical criteria (see 6.5.1).
- **Predict:** The instability of the prototype while interacting with the Unison background process can cause some unreliability.
- **Web interface:** Interaction with the central server's web interface. While not problematical in the persona tests (see 6.4.1.6), it was an issue during development.

- **Exceptions:** Catching and correctly dealing with errors in the Perl programming environment.

These issues will be addressed in the later Summary section in which a revised action plan is created to address these issues (see 6.6.6).

6.5 Technical criteria

This section discusses the technical criteria generated during Section 4.2 of the evaluation criteria. What follows is a breakdown of each of technical evaluation in relation to the initially created Perl-based prototype.

Each of the following criteria is drawn from the initial technical criteria construction detailed in 4.2.1 though 4.2.5.

6.5.1 Distribution

This section reviews the initial prototype against the prototype distribution criteria generated in the corresponding evaluation criteria on prototype distribution, 4.2.1.

Perl was initially selected for its apparent strengths in the area of cross-platform support as per the discussion in the technology selection (see 6.1) and was useful in enabling the prototype to operate seamlessly within different environments and OSs.

Like all interpreted languages, Perl requires an actual executable (in this case, the Perl interpreter run time) to be loaded with the script contents to invoke the actual program. Where Perl can differ in this procedure is in its support of intermediate object code. This object code represents the semi-compiled binary object that is loaded into RAM before the Perl run time executes the semi-compiled program. Because the script pre-compilation stage is omitted, the object code is executed much faster, the interpretation and syntax checking already being done within the first source code to object code conversion.

In addition to the generated object code, Perl also provides a pure machine code compilation system, which requires that the Perl run time be embedded along with the prototype code (either the pure Perl source code or pre-compiled object code). The execution phase invokes a simple decompression routine of this compiled bundle that installs the Perl interpreter to a temporary storage area that then passes the code to the now decompressed interpreter for execution. The side effect of this method is that the program has to be compiled into this intermediate EXE file for each environment (e.g. chipsets, platforms, OSs) and, due to the decompression requirement, is slow to bootstrap.

Compilation into a native Windows EXE to achieve the above execution sequence is accomplished using the Perl2Exe[122], PerlCC[123], PerlPar[124], or Perl DevKit[125] helpers. These allow the prototype to be distributed as a representative EXE file that can be downloaded efficiently over a normal Internet connection as discussed in the prototype representation criteria (see 4.2.1). When executed, the program can begin its larger set up and installation sequence that can be extensively customised

by the programmer. Within the prototype, this includes the display of the installation sequence that can be seen in more detail in the use case walkthroughs.

Since the executable is already pre-packaged, this installation sequence merely takes the form of preparing a suitable installation medium and then copying itself into the appropriate installation destination followed by handing over control to this newly installed copy and then exiting.

In the case of the prototype, the intermediate object code method was chosen as the most efficient on non-Windows environments. If the prototype is placed within such an environment and a Perl binary is present as it is on most Macintosh and Linux-based OSs, that is used instead of the supplied prototype binary for reasons of speed and local optimisation that may not be present in the portable version.

Unfortunately, while Perl does support compilation to a Windows EXE file using the methods described above, the prototype is exceptionally slow to start, sometimes taking up to a minute before becoming fully operational. This is largely because of the decompression and compilation stages Perl has to go though before the prototype is operational. While not detrimental to the application itself, it would be preferable if the prototype could run more efficiently on the Windows platform.

6.5.2 Synchronisation

This section reviews the initial prototype against the synchronisation technical criteria discussed in 4.2.2.

Using Perl without any external modules extremely limits client-server communication by providing only basic socket programming functionality. Its major strength, however, is its reliance on user-contributed modules, which can provide a whole array of functionality outside of the core language. In the context of upload functionality, Perl's *de facto* HTTP library, LWP, provides for extensive handling of the HTTP protocol along with (in the typical Perl style) multiple ways of achieving the same end. Likewise, LWP can easily be connected to other modules such as compression libraries to provide support for GZip download and upload compression.

All of these modules are well documented and provide the cross-platform support that allows the portability goal of the prototype to be accomplished easily and without any wasted effort or reinvention of these concepts.

Encoding standards present for upload and download operations are supported by Perl, which can harness installed pre-compiled executables like GZip or BZip if available on the system. If this functionality is not available, Perl can use its own slower (but native) equivalents to efficiently compress data. This allows optimisations where present but does not negate portability where needed.

Additionally, since LWP forms the core download component, extra modules can be supplied to provide extra functionality. One such example is LWP::Parallel::UserAgent[126], which allows the parallel downloading of multiple data streams without the need to create a custom threaded

application. Modules such as this make the Perl environment one of the strongest languages available for effective prototype creation without having to cover new ground in order to add these extra features to the code base.

File streams to and from a server can be optimised quite efficiently using compression to minimise network overhead. If no pre-compiled compression agent such as GZip or BZip is present (as on Windows platforms, which often lack the libraries to do so), Perl can improvise and use its own built-in solutions. While practical and extremely portable, it is often a detriment to CPU speed since the incoming and outgoing streams are compressed in the slower non-optimised interpreted object code instead of the natively compiled C-generated machine code that may contain many chipset, platform or OS optimisations not available to a more portable compression library.

6.5.3 Package management

The following technical criteria describe user-initiated activities such as package installation and removal as discussed in 4.2.3 of the technical evaluation criteria.

Package atomicity was achieved by downloading pending changes from the server to a temporary directory within the portability environment. Once the data differentials for the package are successfully downloaded, a check is conducted to ensure that no program that requires that data is currently executing. If this situation occurs, the process waits until those processes end; otherwise, the patching and replacement process takes place immediately.

A minor problem with the adoption of the Perl development environment was the requirement that a custom cross-platform solution had to be developed to detect file changes (as initially discussed in 4.2.2). The bug checking and testing of this additional support module was troublesome on various Linux releases since the method for determining executing programs and open file handles can vary between different Linux release branches.

Since the package removal process consists mainly of file-level operations, Perl provides a number of language unique features that allow the fulfilment of this criteria. These include file-level manipulation modules (such as directory tree traversal) as part of its CPAN module DB. While not used directly in the prototype, a number of these modules were used to quickly develop a custom prototype-specific solution. The ability to take an existing Perl module and inspect the code was incredibly useful for quickly picking up the language, as was the ability to quickly bring in other modules to extend base functionality.

Similar to the distribution criteria, Perl's networking and GUI integration provided a number of efficient methods to quickly code the automated upgrade functionality (see 4.2.2).

A minor downside, however, is the process of determining whether a file within the portable environment is in use. Perl regrettably provides no cross-platform method of performing this operation, so a custom solution had to be created. As with most Perl projects, this was readily accomplished using an existing Perl module that monitored file system handles on POSIX-based systems and another module for the Windows environment to interface with the Windows API

and achieve the same effect. These two solutions together allowed the creation of a cross-platform library that could determine the status of the open files and applications regardless of the host OS.

6.5.4 Error handling

This section reviews the initial prototype against the error handling technical criteria discussed in 4.2.4.

As discussed in 6.5.2, the LWP transfer handler provides many diverse methods for detecting and countering errors that may occur during the upload or download process. This includes the ability to fine-tune the number of failed connection re-tries and other fault detection issues when communicating with a remote server.

Because of these features, Perl and the LWP module can correctly detect network disconnections and handle the various networking events that may be subsequently triggered. Along with these abilities, Perl's threading and multitasking libraries such as the above LWP::Parallel::UserAgent[125] module can handle such operations even when spread across multiple CPUs or other system threads.

As with the network disconnection handling, the prototype should also provide sufficient tolerance of hardware level interruptions such as unexpected device removal.

Unfortunately, Perl does not have an established method of detecting media removal. This led to the creation of a cross-platform module to accomplish this task using a combination of Windows NotifyEvent APIs (for the Microsoft Windows platform) and INotify (for POSIX-based systems). While this does work in the test OSs used, this is an environment-specific detection that could possibly have problems within other not-yet-tested environments.

Aside from the requirement to code this functionality, the use of Perl’s existing interaction with the Windows API and POSIX INotify systems provided an adequate basis for this criteria point’s successful implementation.

6.5.5 Development environment

The following technical criteria measure the prototype's performance with regard to the internal development process defined in 4.2.5.

Perl was designed to be a ‘quick-fix’ language, and while it claims support of more long-term design methodologies, it still retains vestiges of this initial philosophy.

The turnaround is quite short for creating a smaller program but can become quite cumbersome for larger projects. Most of these flaws are addressed in the upcoming Perl 6 version release, but a stable release is not yet available.

One language feature in particular, the use of contexts with variables, can frequently cause confusion when passing variables between functions due to its sometimes unclear functionality. An example of this problem would be passing a complex structure such as a function hash.

The language allows simple passing of the hash variable but it does not work as intended when attempting to read back the variable contents. While passing scalars works acceptably, a hash or array must be passed by pointer and then contexted into a local hash variable for re-reading[127]. This confusing language usage has been fixed in Perl offshoot languages such as PHP[128] and Python[129], making this procedure much easier to learn in these derivatives.

While initial adoption of the language presented some problems, a prototype was eventually constructed using the Perl programming language, although the learning time would have been reduced if some of these issues were resolved within the language.

Perl has some issues with the maintainability of code past the initial creation stage. One of the most levied criticisms at the language is its 'write-only' nature, implying that after the initial programming has finished the program, is more or less unmaintainable. Certainly the language's documentation does not dissuade new programmers to follow the principle of another programmer having to maintain another's code, but most of these critiques are levelled at the emphasis on what has to be the language's most notable feature, its regular expression engine.

While not overly used in the prototype, the engine is undoubtedly the driving force behind people wanting to learn the Perl programming language—and, conversely, being put off by it.

The language's emphasis on there being many ways to solve a given problem as well as its sometimes almost puzzling emphasis on Perl-Golf (i.e. as few keyboard *strokes* as possible to reach completion) can make the language obtuse and unapproachable for a maintainer. These issues all add towards sometimes confusing modules, forming an obstacle to proper maintenance or clarity. While some effort has been expanded to eliminate this problem (e.g. commented expressions[130]), the legibility of most internal modules has yet to be improved, which provides a focus for future projects.

While regular expressions are largely credited with Perl's success, the numerous and organised CPAN network[109] representing Perl's own modular package repository must be its other strength. No other language comes close to having such a diverse, freely available and well-maintained modular library, and the community effort behind principles such as test cases (for automating the testing of modules on different platforms) have been hugely influential in the widespread adoption of the language.

The greatest criticism of the Perl-based initial prototype is its lacklustre support for GUI systems, which is covered in further depth in the earlier UI framework requirement (see 6.3.4). The selected Wx GUI system seems very much an incomplete attempt to integrate a graphical system into a traditionally console-based language.

Additional criticism must also be levelled of Perl's poor support for multi-threaded environments. Perl, being ostensibly a Linux-based system, can be disappointing in its emphasis on the process forking methodology at the detriment of the threading methodology for platforms that cannot support the former (Microsoft Windows, most notably) and, thus, suffers further when the GUI

system needs to be maintained within such a paradigm. A selection of Perl projects is attempting to rectify this with the Perl Threads[131] and POE[132] modules, but these are also largely community efforts to supply functionality that should be built in to the language.

In summary, while Perl provides a number of modules, its support for non–POSIX-based systems is lacking, especially with regard to GUI development.

6.6 Prototype evaluation and weaknesses

This section will investigate the success of the prototype against its integration with the framework (see 6.1), technology evaluation criteria (discussed in 6.5) and use case scenarios (6.4).

As with the persona-based testing in 6.4.4, a further threat to validity exists in extending the technical criteria discussed in 6.5 to include more stringent technical criteria. An example of extending the testing done in these two sections would include further research into the application portability requirements in order to test hardware and software limitations (i.e. applications that require low-level hardware communication). While omitted from the technical criteria in 6.5 and the persona level testing in 6.4, the testing of these aspects could form a further research project into such low-level compatibility across OS boundaries. Similarly, the distribution discussion in 6.5.1 could also form the basis of further research into methods of prototype packaging. In both of these suggested cases, it is possible for a future researcher to encounter different results than those outlined due to either different application portability requirements or emerging technologies (e.g. the emulators used for application portability have increased capability).

The initial implementation of the prototype discussed in this chapter used the Perl[133] development environment. The Perl environment provides efficient speed, a relatively simplistic object model and a wide variety of user provided modules via its CPAN distribution channel.

Perl, like most P-code languages (the collective term given to Perl, PHP and Python), is strongly supportive of cross-platform development with the additional advantage that its pre-compiled object code can ensure an almost one-to-one ratio against equivalent C-compiled programs in terms of speed and efficiency.

The Perl-based architecture has advantages over the single-thread model in that the complex RSync implementation is handled exclusively by an external and therefore upgradeable executable (Unison) allowing the majority of the prototype to concentrate instead on the parts of the development that are relevant to the workspace transference framework.

In summary, the key conclusions of the Perl prototype implementation are as follows:

- **Portability:** As disclosed in 6.5.1, the Perl development environment allows for compilation of the prototype into a native executable, but due to its natural state as a scripting language, some problems remain (discussed below).
- **Slow bootstrap:** The use case of the prototype distribution (see 6.5.1) showed that the Perl executable format is actually a self-extracting ZIP archive with an embedded Perl

interpreter. This requirement leads to very slow prototype start-up speeds (known as *bootstrapping*) and large file sizes.

- **Community support:** A large selection of modules provided by the CPAN distribution network allows for efficient code-reuse and sharing (see 6.5.5).
- **GUI integration:** While not as well documented or as easy to use as the Microsoft Forms library (the GUI system for Microsoft based systems such as Visual Basic; see 1.3.1.5 for further details), Wx intends to provide a cross-platform GUI library that operates on the selected test platforms. The Wx GUI framework is a completely different paradigm from the usual GUI implementation methods and, as such, presents a learning curve. Its attempts to implement the functionality of all OSs, while sometimes useful, can also be cumbersome when some of this functionality is unsupported on other platforms.
- **Maintenance:** As discussed in 6.5.5, while some methods exist to prevent code obfuscation, it can sometimes be quite difficult to maintain relatively clean Perl code. As the project size grows, so does the difficulty of future maintenance.

These issues map onto the workspace transference framework requirements and evaluation criteria as shown in Table 8.

Reference	Title	Initial Prototype
	Framework Compliance	
3.1.1	Application portability	✔
3.1.2	Data portability	✔
3.1.3	Data sharing	✔
3.1.4	Environmental interaction	✔
3.1.5	Interface requirements	✘
	Persona use cases	
4.1.1	Novice	✔✘
4.1.2	Knowledge worker	✔✘
4.1.3	Developer	✔✘
	Technical criteria	
4.2.1	Distribution	✘
4.2.2	Synchronisation	✔
4.2.3	Package management	✔
4.2.4	Error handling	✔
4.2.5	Development environment	✔

Table 8. Evaluation criteria pass/fail analysis for the initial prototype

The following sub-sections will discuss these points further and ultimately construct a list of requirements for further iteration of the prototype design and implementation stage.

6.6.1 Prototype portability

This sub-section discusses the portability and bootstrapping issues encountered in the Perl prototype implementation process (see 6.6) and refers to Perl's largely stop-gap solution to the problem of providing a compiled version of interpreted scripts. Compiling Perl executables is a matter of some contention within the community. A compiled Perl binary is effectively a large self-executing ZIP file, which loads a Perl interpreter and extracts all scripts to a temporary directory and then instructs the interpreter to run the scripts as a normal execution process. This process has a number of disadvantages, not the least of which is inefficiency of the Perl interpreter having to effectively re-compile the extracted scripts into memory before they can be executed. Coupled with the extraction time of the interpreter, code and external required modules, this can equal a large delay between the Perl script execution and the actual program execution. Perl is also rather disk hungry with

its modules, and its inter-module reliance means that the number of modules can be quite large. This in turn means more compression leading to more inefficiency, more compilation and a larger executable file. Unfortunately, these issues mean that the multitude of Perl compilers (PAR[123], PerlCC[122], Perl2Exe[134] and DevKit[124]) are all relatively similar and all exhibit huge compilation sizes and slow start-ups.

6.6.2 Slow bootstrap

While relevant only from a usability perspective, the slowness of the prototype to execute when required can be off-putting (see 6.6.1).

The reasons for this are that the Perl EXE container needs to decompress its payload into a temporary storage space before the actual script execution can begin. This slowness only applies within the Windows environment (Linux using a decompression method more suited to this purpose), but the environment in this case, unfortunately, represents a larger user base of potential cases.

Further experiments with various Perl decompression agents all reveal the same fundamental flaws, leading to the conclusion that the slowness of Perl within the Windows environment is unavoidable.

6.6.3 Community support

This sub-section represents the third point in 6.6 and refers to Perl's CPAN distribution network, which provides an efficient and comprehensive method of integrating with existing language modules. Using the CPAN system, it is possible for a developer to quickly integrate an existing Perl module into a project using the Perl programming language or upload his own[135]. In addition to this functionality, CPAN also provides a detailed analysis on module compatibility (via the testing system) across multiple test OS environments as well as an integrated documentation system via its plain old documentation syntax.

Using these features together, Perl demonstrates the strong community basis, one of the central reasons for its candidacy during the technology selection sub-section (see 6.1). While the other points in 6.6 are largely negative, the community and modular support of Perl make it one of the strongest programming environments available today.

6.6.4 GUI integration

This sub-section represents the fourth point in 6.6 and refers to the issues with Wx toolkit[80] implementation and its integration with the prototype. While being a relatively pleasant GUI framework to work with, Wx is sometimes poorly documented. Its insistence on exposing cross-platform methods, events and properties on all platforms can be helpful, but one platform in particular, Microsoft Windows, tends to have poor compatibility adherence. An example of this is the well-supported tree control (Wx class: Wx.TreeCtrl), which implements the standard hierarchical tree widget often seen in file managers to illustrate directory layout. This control provides a number of methods such as AddItem, Sort, AddChild, ExpandChild and other standard tree node functionality. One such event, 'ExpandAll', is a classic example of the Wx attempt to

provide all APIs across all supported Wx OSs but not necessarily with the same expected behaviour on each. The ExpandAll method simply instructs the root of a tree structure (such as a hierarchical display of folders) to expand child nodes. Unfortunately, the underlying Microsoft Forms control (refereed natively on the Windows platform as TreeControl) does not provide for this functionality *unless* the root tree element is visible. This minor caveat is only present on the Microsoft Windows platform and is only discoverable by actually observing this seemingly unpredictable behaviour first hand. Extending this API function to enable the Wx library to work around this issue and present consistent behaviour is a relatively simple expectation, but has not been attempted within the WxPerl release. This platform-specific caveat is one of many such bugs that goes unmentioned in the Wx documentation and can only be found after thorough experimentation of each destination platform. This makes the cross-platform GUI coding exceptionally time consuming as this and other bugs need to be discovered before a project can use them.

6.6.5 Maintenance

This sub-section refers to the fifth point in 6.6 and refers to the difficulty of maintaining Perl programs. While the CPAN network does provide a number of module management methods, the 'ownership' of modules uploaded to CPAN can, conversely, present a limitation if the original module author is either lax in his duties or is unable to provide adequate maintenance. This can lead to large disparities between modules that are well maintained and have excellent documentation, test suites and bug management and modules that fail in one or more of these areas. The status of a module being owned by one and only one developer causes problems should this module author not be able to provide future fixes for discovered bugs.

An example of this phenomenon within the prototype implementation stage was the selection of the configuration storage format for saving user settings. While XML was eventually chosen as a viable storage method, INI files (simple key value configuration formatting) were briefly considered before adoption of the more extendable and universally supported XML format within the prototype. The INI format is a relatively simple example of the nature of CPAN in that it has many possible file format-managing modules such as Config::Abstract::Ini[136], Config::Any::Ini[137], AnyData::Format::Ini[138], OpenPlugin::Config::Ini[139], Config::Backend::INI[140], Config::IniHash[141], Config::INI::Access[142], Config::INI[143] and Config::IniFiles[144]. Leaving aside the sometimes inconsistent naming, Perl clearly revels in the founding principle of ’there’s more than one way to do it’. While the sheer variety of choices to provide support for this relatively simple file format does at first seem beneficial, the inconsistent and sometimes downright erroneous documentation of some modules can present issues for the developer when selecting a single tool for the task at hand.

On a more localised level, the Perl syntax itself can quickly descend into unreadability if standards are not enforced by the project developer. As expected of Perl, these rules are not inherent within the language but rather are placed by a project’s authors upon themselves.

6.6.6 Revised action plan

As outlined in sections 6.6.1 to 6.6.5, Perl represents a good stepping stone language between prototype conceptualisation and an integrated higher-level language, but its cross-platform coding is strongly POSIX-reliant and its inefficiencies with the Windows platform make its long-term use undesirable.

The coding platform is relatively mature, but the testing of modules that can have unpredictable results depending on the host OS can also be a chore.

From the above, we can conclude that the prototyping stage is to be repeated with the following design amendments:

- The compiled executable of the language is to be equivalent of a native OS's program speed, size and efficiency.
- The prototype will be created in a professional working environment with an emphasis on cross-platform portability, especially in areas such as GUI compatibility.
- The prototype will be created in an environment with an emphasis on testing independent system modules.
- The prototype will be created in a language that enforces structural order and maintainable style.

These criteria are implemented in Chapter 7, where the design process seen in Chapter 5 will be repeated with the additional points discussed above.

Chapter 7. Revised Prototype

Following the discussion of the initial prototype implementation (known by its release name WorkTran v1.0) deficiencies in 6.6, it is evident that a further prototype must be commissioned to provide solutions to the problem areas listed in 6.6.6.

This chapter functions as an evaluation of the second evolution of the prototype (released as WorkTran v2.0) in an attempt to resolve these issues. This second prototype was constructed using the prototype design outlined in Chapter 5 as well as the action plan in 6.6.6. As with the initial prototype, the revised prototype will be evaluated against the generated evaluation criteria discussed in Chapter 4. In contrast to the first prototype, this new implementation will use the Python development environment, which provides the requisite solutions. In this chapter, the first Perl-based prototype will be referred to as the *initial prototype*, with this chapter's Python-based system similarly referred to as the *revised prototype*.

7.1 Framework compliance

This section discusses the revised prototype's adherence to the workspace transference framework (see Chapter 3). Sections 7.1.1 to 7.1.5 analyse a series of experiences encountered during the prototype implementation phase applying the workspace transference framework (see Chapter 3), the prototype design (see Chapter 5) and the revised prototype plan (see 6.6.6) against the Python development environment.

7.1.1 Application portability

Python provides built-in functionality for process execution and monitoring without using any additional external libraries. Using these core language features, it is possible to manage processes within the Python environment using a consistent and cross-platform API. This provides the advantage of interaction with both the WINE and CygWin environments via pure console input/ output (I/O) using a stable and well-tested interface.

Additionally, for pipe or socket-based systems, Python provides the choice of two interfaces. The built-in socket set Python supports is extendable, well documented and available on all Python platforms; likewise, the WxSockets library set is as cross-platform as the Wx environment itself. Like all Python modules, it is easily extendable and provides an intuitive method to quickly establish communications to internal processes such as WINE and CygWin.

Unfortunately, Python does not provide a suitable cross-platform method of integrating with application settings. However, utilising Python's cross-platform support and some external libraries, it was possible to construct an API library that communicated either at the file level (for Linux-based systems), daemon interaction (GNOME desktop on Linux) or registry (Windows). These three interaction methods coupled with the Python object orientation interface allowed the prototype to effectively manipulate the local system in order to provide interaction to prototype-managed packages. Python will simply load whatever modules are required on the target system without any additional workarounds. It is a deficiency of the core Python functionality that these features are not

already supported, but management of application settings was easily replicated within the language using pre-existing modules without any notable issues.

To provide portability of the prototype executable itself, Python compiles to native ELF (for Linux) and EXE (Windows) executable formats, which allows for native execution on the supported OSs. Unlike Perl, which favours a mixture of native and compiled object code, Python places a larger emphasis on pure Python libraries, libraries that are themselves written and compiled within the Python language as opposed to being compiled into object or machine code and called on via an API. This method of providing Python-focused libraries equates to increased portability at the slight cost of inefficiency during the script interpretation stage.

Python's ability to compile to a stand-alone executable stands in contrast to the initial Perl prototype in that the bootstrap times during executable launching are radically reduced. This decreased bootstrap time, in addition to the ease of use of the compilation system, is a great advantage to the Python programming environment as opposed to the primarily scripting-based Perl. The technologies used to achieve the prototype portability on each target platform are discussed in further detail in the distribution technical criteria (see 7.3.1).

Using the approaches outlined above, the revised prototype can provide:

- Windows application support within Linux or Macintosh environments
- Linux or Macintosh support within Windows environments
- Platform agnostic support for local OS restrictions
- Hardware-agnostic support
- Interception of OS events and redirection to locally stored resources

The revised prototype also provides the ability to enclose an application within these emulated environments in such a way that any registry or file system operations can be safely stored entirely within a self-contained enclosure. Using this interception method, the revised prototype is capable of redirecting all OS activities (e.g. saving registry entries or registering system components/libraries) to a file system-based storage method, which can then be transported to a similar prototype-controlled environment and reassembled to continue the user session with intact settings. In all of these cases, the prototype demonstrates portability on a scale not achieved by any other research. The mobility provided for applications to seamlessly work on any supported platform provides the user with the ability to potentially mix and match applications as needed from a variety of sources. The wide selection of applications combined with this application portability expands the available application library on any compatible system.

Examples of such application portability are shown during the use case walkthroughs in which the complex OpenOffice suite was successfully transported across OS and hardware boundaries (see 6.4.1.3). In this persona task, the program, AVG virus checker (see 6.4.1.2) and Firefox web browser (see 6.4.2.3) can all be efficiently and transparently moved across hardware or OS

boundaries to the user. All of these demonstration applications successfully redirected system calls and saved the application data to disk to allow transportation along with the portable workspace.

This functionality can be extended to not only packaged prototype functionality but has been shown to operate in a user-created demonstration program (see 6.4.3.4 and 6.4.3.5) where a binary file compiled by the Developer persona within one environment was successfully transported to another machine and OS combination.

7.1.2 Data portability & sharing

The data portability workspace transference framework requirement represents one of the most obviously successful areas of the final prototype, providing remote backup, file versioning (on the server) and package management with little to no user experience required. These features are all demonstrated in the successful persona tasks (see 7.2) and the technical criteria (see 7.3). Data and hardware portability is demonstrated in examples such as moving complex programs across OS boundaries (see 6.4.1.3, 6.4.1.2, 6.4.2.2 and 6.4.3.4).

Data portability within the prototype not only provides an efficient and dependable backup system but also a package update mechanism. Unlike other backup systems that merely attempt to copy data from one location to another, the prototype can understand the dependency between individual files. In this way, relevant data can be updated as one atomic unit rather than as a series of disconnected files. This data portability mechanism is useful when upgrading a packaged application comprising the application binary and dependent resource files. Since the existing application is already installed, downloading and overwriting these existing resource data files causes incompatibility between these separate but conceptually linked files if only some of the relevant data is updated. The prototype instead correctly analyses and updates all of these interlinked dependencies, allowing for the efficient upgrading from one version to another without breaking any file system dependencies between versions.

In order to maintain data integrity, the prototype utilises a versioning system that allows the user to effectively ‘rewind’ a file to any potential point within its lifetime as shown in the data corruption use case task (see 6.4.1.5). Using this method, the prototype provides a data integrity layer and allows the reversal of any potential file action by either the user or the remote prototype server. One such example would be the merging of two distinct file versions. This operation can occur when the prototype is in use on two or more machines. Under this scenario, two distinct and potentially contradictory states of the same file can be present. With the addition of versioning, the server can safely attempt the merging process between file types. Depending on file type, this can be relatively easy (e.g. plain text or source code merging) or difficult (e.g. merging proprietary binary blobs or proprietary document formats). Most code management suites have implemented file merging in this way for many years and the prototype can harness existing work in this area for this use.

The versioning and merging system allows the user to potentially work on various file formats on multiple machines or even share resources with others with the knowledge that the file can (in most cases) be successfully managed between the individual prototype nodes. There are, however,

compromises in this system; for example, the complexity of some formats means that this is not always a successful process. If in doubt, the user can take the two file states and merge them manually. This situation should lessen over time as more and more merging systems are added to the server, allowing for better understanding of the conflicting files.

The duplication of files for the user to resolve manually is largely a stop-gap solution designed to prevent data-loss. The ideal scenario would be the automatic resolution of conflicts internally within the prototype's data portability layer. Unfortunately some file types are intrinsicly difficult to merge (e.g. binary data, pictures, databases) and therefore must be resolved by the user. The topic of binary merging is not centeral to this thesis or to the research question and, while not ideal, the solution detailed above should satisfy most conflict scenarios.

A potential limitation here lies with the packages managed by the prototype. Not all software packages are capable of working with real-time updates, and some may require closing and re-opening of the altered data stream before online changes are reflected. This is most commonly displayed in legacy word processing software where the file being edited can be changed on disk (in this case automatically by the prototype in sync with external changes) but the file needs to be re-opened by the software in order to reflect the new changes.

As with the initial prototype, synchronisation was achieved in the revised prototype by using the Unison project as a worker module. This design decision stems from the conclusions reached in the initial prototype (see 6.3.1) that an external project must be used to perform the synchronisation operations. Further breakdown of logical operations for the Unison module along with its integration within the prototype can be found in the Appendix (see 1.2.5).

The prototype's responsibilities with regard to data portability are to keep local files in sync with the remote server and to translate environmental settings into a storable data format so they can be synchronised and manipulated by the server as necessary. By treating all resources as data files, the prototype can successfully perform a number of actions including versioning, conversion and allowing for future API integration.

By using Unison within the revised prototype, the synchronisation operations can be outsourced to the external Unison module. The prototype can draw on the existing work done in external projects such as Unison itself, the RSync project (the protocol Unison uses for transport) and SSH (the secure connection method used by the RSync component within Unison). By leveraging these technologies, the prototype can instead specialise its core portability functionality by mediating the transfer operations at a higher level of abstraction.

Python does provide an existing cross-platform process manager under its ‘sub-process’ libraries, but these are not completely cross-platform due to the inherent differences between the major OSs. In order to combat this deficiency as with the previous Perl prototype, a wrapper module was created (called PyPredict, see the Perl equivalent in 6.3.1) to provide a dependable single API to control the Unison sub-process regardless of OS.

After creation of the Predict module, cross-platform IPC was accomplished within the revised prototype to the satisfactory levels used in the initial prototype.

In conclusion, the prototype provides data sharing to a limited degree, the very nature of file-based systems puts client-based systems at a disadvantage to offerings such as Google Apps or Lotus Live, where multiple people can work on the same ‘file’ simultaneously (file here being a conceptual term instead of a representative of disk-stored data). The primary reason behind the compatibility with web-based solutions is that these newer solutions have been designed with data sharing in mind rather than as an afterthought as the technology evolved.

The prototype does provide the ability to share the same file provided that the server understands the format and the client editing software is capable of reflecting changes made as the altered data is updated. If this is not possible, the server is capable of storing the conflicting file states and allowing the user to manually perform the merge operation.

7.1.3 Environmental Interaction

Python has good cross-platform support but its major deficiency in comparison with Perl is the lack of any main distribution point for information. Python would be improved immensely by the addition of a CPAN-like system that would allow both distribution of modules and extensive testing. Instead Python modules are provided by the individual coders in a variety of inconstant ways. This method is not uncommon to most languages, but like all projects that lack an established distribution system, these scripts lack quality control and can be buggy and lack documentation or any kind of compatibility information. Python’s reliance on decentralisation stands in stark comparison to Perl, with its emphasis on off-the-shelf integration with other established and well-tested projects.

Both the initial and the revised prototypes are compatible with most file systems that the prototype suitable for the supporting OSs targets. With the assistance of some libraries to handle these file system incompatibilities, the file systems are transparently mapped as needed. In order to simplify internal file operations, all internal file-related tasks are conducted in the UNIX format (e.g. '/dir1/dir/dir3'). This is automatically translated into the host platforms path format whenever file access occurs. The only caveat to this rule is the lack of any consistent method within both Perl and Python to reliably determine the location of the document directory (e.g. 'My Documents' on Windows, '/home' in Linux, see pd-3.1.4 for further details). These issues can be negated with workarounds, but it is disappointing that there is no official implementation of this functionality within the language’s core module set-up.

Python correctly imports most environmental details from local settings such as system variables (Linux) and common registry details (Windows). Like Perl, more complex entities such as networking details have to be individually queried from their individual data stores.

Python’s strong object orientation helps immensely with these implementation factors, allowing the creation of a strong object architecture that can be easy extended depending on the OS or active environmental details.

Both of the major emulation projects used within the prototype, CygWin and WINE, provide facilities to work with the host OS resources and provide their own emulation layer, should it be needed, to any packaged application. Using this system, an application could run natively (e.g. a Windows application can run under a Windows host) or in an emulated mode (e.g. the same Windows application within WINE on a Linux host). These emulation systems can also provide prototype-managed interaction with the host OS, allowing integration with GUI, disk, printer and network resources.

CygWin maps most of the OS transparently to its own internal UNIX standard (i.e. transforming hard disks into UNIX block devices). WINE likewise creates its own self-contained environment but adds the host machine's root directory as one of the 26 available letter drives ('Z' by default) for access in the same way.

Most hardware available to the local user is immediately made available to the prototype provided that the emulation layer has the appropriate knowledge and drivers to access it. For example, printers need to provide either Postscript or PCL compatibility for CygWin in order for any emulated Linux software to be able to print.

An exception to this rule is networked printers, which do suffer from some impedance for both CygWin and WINE. The main problem area is that specialty drivers are often used to control these devices that are difficult to access on Linux (via CUPS) and Windows (via the Windows Common Internet File System layer) systems. For the most part, this problem tends to be unidirectional—mainly CygWin printing on Windows—due to the CUPS layer's difficulty in communicating to no PostScript devices. This problem is slowly being solved with the addition of PCL and other less used formats.

Use of the OpenOffice word processor in 6.4.1.4 demonstrates the interaction of this prototype-controlled package from the application itself down to the file system that is used to save the file to disk.

In addition to transporting OS-bound settings, the prototype can also provide for changes between environments.

The best illustration of built-in environmental settings portability can be seen in web browsers or other Internet connected applications that store proxy information within their own configuration. This presents a problem when these applications are transported between environments necessitating either manual intervention to change these relevant details on the new host or the construction of a more complex system to 'translate' the application-specific configuration settings during the transfer process. Updated versions of some applications (e.g. the Firefox web browser) now have the option of determining per-environment proxy information from other programs (e.g. from Internet Explorer on Windows systems) or other global system settings (e.g. the HTTP_PROXY variable on Linux-based systems). While this does reduce these environmental problems to some degree, it does not completely eliminate them for more complex or less forward-thinking applications.

User level settings are transported along with the other file data belonging to the user and thus can also be tracked within the server's versioning system. However, conversion of these settings requires some effort between machines. For example, the Windows registry is intended to be a central storage point for user and system settings. The prototype can provide portability here by transporting these setting between machines and providing selective alterations for host-dependent settings (as in the above example with proxy information). This applies only to packages held within the prototype that can safely be removed at the end of the prototype session. Likewise, various packages can be installed that provide certain other effects to registry settings depending on the host, such as switching the mouse button (for left-handed users) or changing various theme or visual details such as the desktop background to the user's own tastes. As with all non-essential services, these are relegated to optional packages that the user can select to install as desired. The use of environmental interaction with the prototype can be seen in 6.4.2.3, where the Firefox web browser was reconfigured by the prototype to the correct host-derived connection details. This integration removed the needless complexity of specifying these options for each application on each host.

The prototype can successfully translate these application settings between Windows (mainly registry-based), Linux (mainly file-based) and Mac OS (a mixture of the two methods) and provide the same functionality potentially on any machine to which the prototype is transported.

7.1.4 Interface requirements

Unlike the earlier Perl prototype, Python's bindings to the Wx library (via WxPython[134]) were well documented and clearly defined. Generally, the consensus regarding the initial prototype's Perl language base is that a GUI is generally considered an optional and largely unnecessary element of computer program design. Perl's focus is instead on console-based interaction, an attitude reflected in its largely incomplete and buggy Wx GUI implementation.

Python, however, presents a clear and well-supported multi-platform Wx implementation, which solves the deficiencies of the initial prototype implementation deficiencies in a number of ways:

- **Easier installation:** Installation and configuration of the Wx library suite involves far fewer stages within the Python environment. Unlike Perl, the process for which requires a large number of intermediate compilation stages for the active platform, WxPython is readily available from the package repositories on most Linux systems as well as its availability as a stand-alone installer for Windows machines. Using these distribution channels, the WxPython environment set-up was far more efficient than the method employed in the initial prototype.
- **Better documentation:** Documentation available via the Perl repositories is poor and incomplete. Most available examples are dated and apply only to now-defunct versions of the Wx API. The WxPython project, however, has provided a central repository of Wx GUI information and encourages a community effort for documentation contributions, code examples and, most importantly, the organised bug checking of Wx functionality.
- **Rigid interface:** As with the Python philosophy ('there's more than one way to do it'[145]), rather than attempting to provide numerous ways of solving the issue of GUI integration (as with Perl[146]), Python instead promotes single-path integration of the Wx library

system. Using this single approach to integration, the revised Python prototype does not present some of the cross-platform issues present in the earlier Perl development.

- **Compatibility with WxGlade:** As with the previous Perl-based prototype, WxGlade was used to generate the GUI interfaces within the prototype. Since WxGlade can switch output formats from generated Perl code to Python code, the initial designs within the Perl project could be easily ported into the Python project with no additional implementation time spent on the GUI design process.

Due to these factors, the WxPython GUI implementation was much cleaner and more efficient in the revised prototype than in the previous Perl-based solution, with less overhead expended towards the issues detailed in 6.3.4.

The prototype, using its default GUI system, represents a simple point-and-click–based method for efficient package management. The use case walkthroughs demonstrated the success of this by showing that all three selected personas could interact with the prototype during its initial installation (see 6.4.1.1), regular use (see 6.4.2.3) and even in more restricted environments such as a GUI-less, terminal-only interface (see 6.4.3.5).

During expected and regular usage the prototype is essentially a transparent daemon running the background of the user's workspace, silently synchronising files to a central server. When instructed to perform an action (typically by the user clicking on the prototype's system tray icon), the prototype presents a Wx-based GUI system that guides the user though the action he wishes to perform, normally package installation or removal or file restoration from some past snapshot.

The Wx system provides a cross-platform solution that is standardised in its general outlook for all host OSs, thus the same client run on Windows-, Linux- or Macintosh-based systems always maintains the same interface whilst using the local OS style and accessibility methods.

Disconnecting the prototype from an Internet connection obviously has a certain number of drawbacks, namely that the files written to the prototype's managed storage area cannot be backed up online or versioned, distributed or maintained. While this is not a major problem in itself, it is inadvisable for the user to disconnect from the Internet for prolonged periods of time should the medium that the files are held on (e.g. a USB disk) become damaged, resulting in data loss to the point of the last online synchronisation.

Assuming that the risks are necessary in a specific context (e.g. working on an airplane where it is desirable to use the prototype without Internet connectivity), the prototype would work correctly regardless of a suitable Internet connection being available. This is demonstrated in the 6.4.3.3 use case task in which the prototype was purposely disconnected from an active Internet connection in order to test its stability under such conditions.

As a subset of this category, the prototype also works correctly though any local networking restrictions such as VPN, tunnel or other such networking logical present on the host system’s routing tables.

7.1.5 Framework compliance summary

While Perl has the highest availability for external modules, Python's general community feel and ease of use make it a suitable medium for cross-platform compatibility. Its efficiency is acceptable for an interpreted language and its stand-alone compilers are far ahead of Perl's, points that make the Python language ideal for cross-platform distribution.

Python is the strongest candidate prototype and represents the most compatible solution with respect to cross-platform development. Should a centralised module system be implemented in the future, Python surpasses Perl in most of the examined workspace transference framework points.

The above sections have demonstrated that the revised prototype successfully satisfies each of the workspace transference framework requirements.

Some minor exceptions to this rule include package or third-party applications that are incompatible with some framework items, the most prominent of which is the data sharing requirement. The gap with this implementation is the need for a packaged application to check the underlying file on disk for changes in order to verify the integrity of any currently active files. Some larger applications, including most office suite software and other proprietary applications, are especially susceptible to this deficiency. The lack of ability can be disadvantageous if a user has to manually trigger a data refresh by closing and re-opening the file to reload it. For the most part, FOSS software, plain-text editors and some more advanced integrated development environment (IDE) systems all provide this behaviour and successfully enable the user's ability to potentially work on the same file on separate machines.

This problem can be considered minor for most of the user base. While more experienced users are likely to desire this behaviour, it is not usually requested by the less able user base.

As such, in all major areas except the above, the prototype successfully demonstrates the practicality and real-world viability of the workspace transference framework.

7.2 Persona use cases

Since the initial and revised prototypes use the same GUI frontend system, the results of the persona tests on the UI aspect of the revised prototype returned the same results as those used in the initial prototype (see 6.4).

As discussed in the conclusion of the initial prototype, the implementation problems with the initial prototype were largely related to support and testing within different OSs rather than any GUI-related issues. The change of implementation environments from Perl to Python has resolved the issues discussed in 6.4, but since these issues mainly focus on lower-level functionality (e.g. maintenance concerns of the Perl-based prototype), the overall appearance and operation of the prototype was largely unaffected from the persona testing perspective. No additional issues were introduced to the persona tasks during the revised prototype implementation stage.

Chapter	Persona Use Case	Issue	Issue Section
6.4.1	**Novice persona**		
6.4.1.1	Install the prototype on a desktop machine	Bootstrap	7.2.1
6.4.1.2	Install a free virus checker package	Predict	7.2.2
6.4.1.3	Install a word-processor package	Predict	7.2.2
6.4.1.4	Type and save a simple letter		
6.4.1.5	Change the document contents	Predict	7.2.2
6.4.1.6	Attempt to recover the first version of the file	Web interface	7.2.3
6.4.2	**Knowledge Worker**		
6.4.2.1	Install the prototype on a netbook machine	Bootstrap	7.2.1
6.4.2.2	Install a web browser package	Predict	7.2.2
6.4.2.3	Use the web browser to bring up a web page		
6.4.2.4	Install the prototype on a desktop machine	Bootstrap	7.2.1
6.4.2.5	Open the web browser and view history	Predict	7.2.2
6.4.3	**Developer**		
6.4.3.1	Install prototype on a desktop system	Bootstrap	7.2.1
6.4.3.2	Install GVim and the Perl scripting language	Predict	7.2.2
6.4.3.3	Disconnect from the Internet	Exceptions	7.2.4
6.4.3.4	Compile a simple script on Linux		
6.4.3.5	Demonstrate the application on Windows	Predict	7.2.2
6.4.3.6	Install the prototype on a remote Linux server	Bootstrap	7.2.1
6.4.3.7	Create a package from the application		

Table 9. Issues uncovered during the initial prototype implementation

Table 9 (which is a copy of Table 7 reproduced above for reference) shows the breakdown of issues encountered during the initial prototype persona evaluation stage. Since the overall appearance and functionality to the user is identical between the initial and revised prototype, the actual prototype persona testing stages will not be repeated here. Instead, Table 9 will be discussed in relation to the issues solved during construction of the revised prototype.

Some of these issues, such as the bootstrap time (present in all install operations) and the uses of PyPredict (used in all synchronisation tasks), are duplicated for similarly related tasks. Others, such as the use of a web interface for server-based interaction and the use of Python's exception handling, only occur in one specific persona task.

7.2.1 Bootstrap time

As discussed in the Technical Criteria section (see 7.3.1), the initial prototype's Perl environment provides a pre-compiled distribution method for the initial prototype, but the executable's start-up times are exceptionally lengthy.

These slowdowns are particularly evident during the initial start-up of the prototype on a new system where all tasks involving the prototype execution in order to set up a new portable workspace

display this behaviour. Regrettably, changing the Perl compilation method does little to lessen the bootstrapping process as detailed in 6.5.1.

It should be noted that the initial prototype does function as expected, but the extremely slow prototype start-up times can cause some confusion in users who may be unsure as to whether the prototype functions correctly. Executing the prototype in the normal way (e.g. by double-clicking) is followed by a measurable wait of up to one minute before the prototype's system tray icon appears in the icon bar. As shown in Table 7, this bootstrapping issue detrimentally affects five of the persona tasks and causes an unfavourable impression on any tested persona.

The revised prototype dramatically improves upon the bootstrap time by providing a responsive and immediately available interface to the prototype. The technical reasons behind this issue's resolution can be found in the distribution technical criteria (see 7.3.1).

7.2.2 PyPredict

The use of PyPredict within the revised prototype provided a number of advantages over the previous Perl-based Predict system. As Python provided a better process control architecture, the stability problems present in the Predict system with the Perl environment were largely eliminated in the PyPredict system, eliminating the crashes and unpredictable behaviour from the sub-processes (specifically the Unison synchroniser used in the prototype) and resulting in fewer crashes of the prototype software.

A technical discussion of the PyPredict module can be found in the synchronisation technical criteria (see 7.3.2).

7.2.3 Web interface

While not the main concern of this research, the server interface provided by the two prototypes was a minor factor during the persona testing phase. For each of the two prototypes, in order to maximise the use of each programming environment, the same language used to create the client-side prototype was used to implement the server-side components. During the prototype testing stages, the server component is largely transparent except for the file recovery (see the evaluation criteria defined in 6.4.1.6).

In addition to the client-side functionality of the revised prototype, the Python programming environment provides the Django web framework[107]. This framework allows for efficient creation of server-side components that can complement the client-side functions of systems such as the revised prototype.

The initial prototype provided a similar system in the form of the Mason[108] server-side framework. Within the context of the prototype, Mason did provide a rudimentary interface with which the initial prototype could interact. Mason's emphasis, however, is on stability and scalability, the latter of which does not suit the nature of prototyping. Django provided a clean, efficient and user-friendly prototype web interface to accompany the Python-based revised prototype.

From the persona testing perspective, Django improved upon the basic Mason framework by providing a much more visually appealing and rounded server interface.

7.2.4 Exception handling

From the perspective of the underlying languages, both Perl and Python provide methods of counteracting the exceptions raised during prototype execution. This was especially noticeable during the error detecting tests such as the Internet disconnection task detailed in 6.4.3.3. During the task persona testing stage, Perl could counteract most of the events given to it by erroneous data connections but was still susceptible to a crash from lower level modules that would fatally exit if the data or device connection were interrupted. The revised Python prototype could take advantage of the much more mature and widely used exception handling functionality of the Python language, which could correctly manage these issues.

Like the use of the PyPredict module (see 7.2.2), the program crashes exhibited by the initial Perl-based prototype do not affect the UI to a great degree. However, the elimination of such program instability ensured that the revised prototype was more usable to all three tested personas.

7.2.5 Persona use case summary

As sections 7.2.1 to 7.2.4 demonstrate, Python improved upon the issues raised during the persona testing conducted in the initial prototype (see the persona specification in 6.4).

These issues were largely due to the improvements within the Python language itself. Python's improved exception handling eliminated many underlying stability issues that caused problems with the initial Perl project. Likewise, the more efficient use of the executable compilers available to Python eliminated the long bootstrap times of the initial prototype.

Additional Python projects, such as the Django web framework, helped the development of a suitable prototypical web server interface that could be used by the personas to restore files. Python's improved modulation and process control allowed for creation of the PyPredict project to control the Unison synchroniser.

7.3 Technical criteria

This section discusses the technical criteria generated in 4.2 in relation to the revised prototype. What follows is a breakdown of each of the technical evaluation criteria in relation to the revised prototype created in the Python development environment.

Each of the following criteria is drawn from the technical criteria detailed in 4.2.1 though 4.2.5.

7.3.1 Distribution

This section reviews the revised prototype against the prototype distribution criteria generated in the prototype distribution evaluation criteria (see 4.2.1).

Python, like Perl, shows strength with its cross-platform mobility. Where Python excels over Perl, however, is with its support of compiled executables. Python compiles executable binaries using the same method employed by Perl, namely creating a self-contained compressed archive with an automatic decompression agent in the header. This complication is accomplished using either the PyInstaller[147] or Py2Exe[148] projects, which can create these stand-alone Python executables on a variety of OSs along with further optimisation such as the Psyco project[149] for Windows-based systems. After decompression of the Python interpreter, the bootstrapper.py script file is read and the execution continues much the same way as if a local Python installation had been invoked with a specified .py file. The Windows executable format can be further compressed using the standard Ultimate Packer for eXecutables (UPX)[150] compression scheme that offers further size savings at a minor cost to start-up speed.

While somewhat cumbersome, this means that a Python file is interpreted in much the same way within the binary as if it were on any other machine. The small and organised nature of the Python core also means that bootstrap time (the time it takes for the code base to uncompress and run) is much shorter compared to the equivalent Perl process. Adding to the speed of the initial program's bootstrap is Python's native support for .pyc (Python compiled) files. These represent the object code loaded into the Perl interpreter and can make the initial load period of any Python module much faster when used instead of the default plain text .py files.

Using a combination of .pyc and compiled Python binaries, the prototype can be represented as a single executable file, enabling easy transportation and downloading for any potential user.

The above shares similarities with the Perl compilation system but is helped by Python's use of pre-compiled Python scripts (.pyc) that do not require disk presence before execution. This stipulation is present within Perl-compiled binaries and is the reason for its overly long bootstrap time when launching a compiled program (see 6.5.1). The use of these pre-compiled binary objects at the interpreter level means that the optimised code can be passed onto the Python VM without the length decompression needed by Perl scripts. By employing these methods, Python programs enjoy an almost instant bootstrap time on modern hardware, which is a measurable improvement over the more cumbersome initial Perl prototype.

7.3.2 Synchronisation

This section assesses the revised prototype using the synchronisation technical criteria discussed in 4.2.2.

Python has a different philosophy from Perl in that while Perl encourages the attitude of 'there's more than one way to do it', Python attempts to consolidate all mechanisms into a single unified library ('there is only one way'). Following this principle, Python's single HTTP networking library, UrlLib2[151], is extensively tested and debugged, allowing for a large number of pluggable user-contributed components.

These pluggable modules change the behaviour of various sections of the main core library using the object orientation approach employed by Python. An example would be using a module to connect to a server though a proxy system. The default UrlLib2 module is initialised and a second proxy module is then added to the library's main core, which changes the module's behaviour to make proxy-based requests. This plug-in–like architecture allows for relatively easy module-based additions as and when required and theoretically ensures that the module is relatively future-proof since the core functionality remains the same with additional modules providing extended support where needed. The use of UrlLib2 contrasts with Perl, where the typical approach when presented with missing functionality is to fork the existing code base and create another similar module with the requisite functionality. While the Perl philosophy generally means a large selection of modules that can perform a set function, Python instead insists on one module that performs all functions. This is a central philosophical difference between the two development environments that is repeated in most other areas including file system access, IPC and other functionality used within the prototype implementation.

The open API standard of these Python modules ensures that potentially any variation can be quickly built into the core networking library. Using this system, compression is easily added by importing the compression functionality from the associated libraries and adding it to the main UrlLib2 core system via its extendable interfaces.

The initial experience with UrlLib2[151] was one of minor frustration due to lack of documentation for all but its most basic functionality. The Python Software Foundation has provided all of the necessary tools for common tasks such as proxy compatibility, authentication, HTTPS and error processing, all of which are needed in this prototype, but have failed to provide usage examples on how to provide necessary hooks into each module's functionality. For example, the method to provide the user name and password for a basic HTTP authentication system is completely omitted from the official documentation. Some of the relevant documentation has recently been improved upon but the initial lack of documentation for this and other core modules has been a constant problem throughout the prototype development process and certainly contrasts negatively against the Perl CPAN system. Instead, most real-world examples of the UrlLib2 library were drawn from *UrlLib2—The Missing Manual*[152], which attempts to address these and other deficiencies in the Python documentation.

Like uploading, Python attempts to enforce its overriding philosophy of using a single strong library rather than multiple smaller components. Thus, downloading has much the same structure as is found in the uploading system. The UrlLib2 library's module layout also allows for compression standards such as GZip and BZip to be built into the networking library with relative ease. This functionality allowed inline translation of whole data streams to and from the remote server, providing large savings via data compression. The cross-platform nature of these modules allowed for easy integration on all Python-supported OSs.

Similar to Perl, the Python networking system is binary-safe and allows for omission of the previous Base64 encoding workaround (see the scoping prototype's need for this in 1.3). Additional

differential upload compression and RSync algorithm implementation also reduces upload bandwidth and ensures efficient transfer into the versioning system on the server.

Full support for UTF8 character sets at the compiler level (compared to Perl, which has only partial support) allows for correct translation between OS language standards and eliminates the need for additional encoding mechanisms of the uploaded data.

7.3.3 Package management

This section describes user-initiated activities such as package installation and removal as defined in 4.2.3 of the technical evaluation criteria.

Importing of the GUI system from the initial prototype meant that little additional coding was needed to perfect the installation sequence within Python. The WxPython[134] interface is compatible with the existing layout and, in fact, fixes some of the bugs within the Wx Perl implementation (see 6.3.4).

Because of the simple transferability of the existing GUI design and the Python Wx implementation, the GUI worked as expected with no additional effort being expended on translation between the Perl and Python prototypes.

One minor criticism of the Python language, however, is its lack of a concise and usable XML interface. The language has no officially supported XML interface, and its large number of XML processing libraries can at first seem overwhelming. XML-TRAMP[153] was eventually selected since it provided functionality similar to the familiar Perl XML::Simple[154] library but with the use of Python's object architecture rather than the nested array and hash structures. The use of this module represented a minor learning curve, mainly because of its hobbyist background and less than full support for most features unlike Perl's numerous advanced XML processors. However, Python's introspection functionality coupled with the use of this module does provide a relatively clean and efficient method for accessing XML files—in this case server communications—to install modules.

The actual file interaction required to install packages within the Python prototype was clean and efficient, and Python easily provided some of the cleanest and most well documented cross-platform file manipulation libraries.

As with the installation sequence, client-side file removal is a relatively straightforward operation. Python's data structures, including file tree traversal, is provided with the use of built-in modules to allow simple recursion operations on a hierarchical file structure. The integrated nature of these operations provides cross-platform support and compatibility with all hardware that is capable of running the Python interpreter.

Additional use of the Python error catching and provisioning functionality can also ensure the transaction-like basis for clean and complete package removal.

Like Perl before it, Python does not provide any cross-platform method of detecting file usage across OSs. This area is slowly improving with the addition of some newer file object classes, but it is still not at a usable standard at the time of development.

In order to combat the problem of cross-platform compatibility, a modular multiplexer system was implemented to conduct checks of the file system use and monitor the appropriate OS interface. Although it is functional on all major OSs (specifically Windows, Linux and Mac), this unfortunately means that the other non-standard OSs also require the appropriate modular functionality. Should this become an issue in the future, the possibility of per-OS plug-in support would allow for future extendibility.

As with the prototypes before it, there is no concise cross-platform solution to the problem of program and file-system use detection. As with most other operations, though, Python creates an OS-level multiplexer to delegate this task to the correct per-OS module relatively easily.

As with the first prototype, the process of maintaining atomicity was accomplished by downloading pending files or by patching differences to a temporary storage directory and checking the target file's use before replacing them. In this way, the contents of a package can be replaced by the newly upgraded files (and/or applications) without any partial states in between.

The file-based process above was easily accomplished with Python, but as with the previous prototype, a number of workarounds had to be implemented to correctly replicate the functionality on Linux platforms. Python correctly detects file locks on the Windows OS but the multitude of ways of implementing the similar behaviour on the Linux OS is not completely supported and, thus, must be replicated within the prototype before it can be used.

Python provides no support for the detection of running programs but as with the previous prototype, the simple addition of a multiplexer and the inclusion of relevant modules for each OS was easily accomplished and successfully integrated to support this detection.

In summary, Python provided some existing cross-platform compatibility much more efficiently than the previous prototype's Perl programming environment, but both environments lack functionality to detect executing processes.

7.3.4 Error handling

This section reviews the revised prototype against the error handling technical criteria discussed in 4.2.4. It also discusses the issues regarding network disruption and disconnection as well as storage medium disconnection from the host machine. These areas represent the common functionality of the prototype in its client–server communication and perform the backup of local resources, which doubles as the package installation and updating mechanism.

The UrlLib2 library's modular interface (see 7.3.2) already allows for efficient handling of interrupted network connections. Additional modules that also allow for network roaming such as the disconnection from one Internet access method followed by reconnection via another, a

frequent occurrence with multiple access points, can be added to the UrlLib2 interface to allow for transparent, error-free data transfers. Unexpected errors can also be trapped and handled via the Python Try/Catch error countering system that is fundamental to the Python language. Together, these features equip Python to deal with any number of potential pitfalls when detecting and countering network disconnection issues.

The approach taken by Python to detect media disconnection is the polling method whereby Python periodically checks for the existence of the data storage device that the prototype is using as a workspace. Unfortunately, like Perl, it had no ready-made solution for implementing a cross-platform version of this behaviour, which led to the development of a custom implementation specifically for this project.

Some partially existing methods conduct the background work on this disconnection detection. One such project for UNIX-based systems, *PyINotify*[155], provides a suitable kernel integrated callback routine for file system changes within POSIX systems. Similarly, Windows exposes functionality with its NotifyEvent API collection that performs device event detection on Windows platforms. Combining these two services via a multiplexer module was a relatively straightforward task within Python that allowed cross-platform support via a consistent internal API.

Using these two systems, it is possible to capture and detect all file-level changes, file system detection and media storage disconnection. By implementing the above libraries, the revised prototype could efficiently handle all forms of service disruption that may occur during regular use of the software.

7.3.5 Development environment

The following technical criteria measure the prototype's performance with regard to the internal development process defined in 4.2.5. These criteria are designed to form an overview of the implementation stage and any issues encountered with the prototype of the development iteration.

Python presents itself as a moderately simple language syntax that layers its complexity into various outer modules. Its emphasis on this object-orientated platform places it at odds with the more complex Perl language, but this is ultimately for the better. Python is easier to learn, quicker to deploy and is becoming the language of choice in larger projects due to its emphasis on object orientation. Unlike Perl, Python presents less complexity in its syntax and makes no attempt to be terse in favour of clarity. With these features, Python enabled the quick and efficient creation of a prototype and allowed deployment to multiple platforms with little effort.

Python places great emphasis on the concepts of code reuse and object orientation; thus, it presents a very strong framework to create a project as maintainable as the produced prototype. The final code generated is split off into well-maintained and documented modules (via Python’s own introspection schema) and can easily be edited by any moderate Python programmer.

In addition, variance with the Perl community with Python’s principle of 'there is only one way to do it' should theoretically mean that two programmers working on the same problem should generate

the same code. This, of course, is not viable in practice, but it means that a lot of the indecision surrounding half-completed modules, a commonplace theme with Perl, is not present in Python.

While not as efficient as Perl during its run time, the relatively clean syntax of the Python language means that source code compilation is relatively efficient. Perl has to not only compile its code but also determine the method by which variables and functions are called (Perl refers to this as *context*); thus, it has to go through additional complicated steps to arrive at a compiled byte code. Python also resembles C++, making some Python scripts natively transferable into that language. This alone is assisted with the PyPy project[156] attempts to study the Python language itself in order to better optimise its own concepts. With such an effort being placed on Python's efficiency as a language, attempts are being made that, in the future, could rival the currently stronger Perl project in its native efficiency. Although not yet to the standard of Perl and the CPAN network, Python presents a friendly community that assists in the development of Python modules and helps newcomers to the language.

Interestingly, Perl has a more rigid discipline in its community than does Python. Perl presents a single point of connection, CPAN, which acts as the documentation and distribution hub for Perl modules. In contrast, Python presents no such central point and instead relies on the standard Internet-based approach of lots of smaller specialist communities. Based on the already stated goals of the two groups, it can be somewhat puzzling that they have a community spirit at variance with their underlying philosophies. Nonetheless, the recent releases of the Python core project have breathed new life into the various Python communities along with the various ports of Python to other environments such as .NET (via the IronPython project[157]).

7.4 Prototype Evaluation

This section will explore the conclusions reached during the creation of both the initial and revised prototype and determine the compatibility of each with the workspace transference framework and the evaluation criteria.

Reference	Title	Initial Prototype	Revised Prototype
	Framework Compliance		
3.1.1	Application portability	✔	✔
3.1.2	Data portability	✔	✔
3.1.3	Data sharing	✔	✔
3.1.4	Environmental interaction	✔	✔
3.1.5	Interface requirements	✘	✔
	Persona use cases		
4.1.1	Novice	✔✘	✔
4.1.2	Knowledge Worker	✔✘	✔
4.1.3	Developer	✔✘	✔
	Technical Criteria		
4.2.1	Distribution	✘	✔
4.2.2	Synchronisation	✔	✔
4.2.3	Package management	✔	✔
4.2.4	Error handling	✔	✔
4.2.5	Development environment	✔	✔

Table 10. Evaluation criteria pass/fail analysis

As shown in Table 10, the revised prototype improved upon the initial prototype and demonstrated success in the areas that were lacking in the initial prototype.

After some research into executable speeds and ease of use, Python[158] was selected as the revised implementation environment for the prototype. Most of Perl's functionality is directly transferable to the Python environment, specifically the Wx coding environment[134], and nothing was lost to impedance between Perl to Python. Python's naturally object-orientated interface is much more adaptable to the Wx paradigm than the older and less object-centric Perl.

Python's stronger emphasis on object orientation and its design testing tools provide the much needed module independence required to separately test the workspace transference framework units. Due to this and the extremely modularised language syntax, it is possible to develop each module of the framework in complete isolation from the others, leading to a lower build time. Its compilation system is slightly similar to the Perl method, however, with the added advantage that Python places less of an emphasis on on-the-fly code compilation. Thus, the object code generated

by Python can be transferred between Python interpreters in an optimised and pre-compiled form with no additional conversion process required.

This implementation cycle had the following conclusions:

- **Wx GUI design:** Wx presented numerous compatibility problems within the initial Perl prototype, and these issues have been addressed by the much more rounded Python implementation. The WxPython implementation provides wrappers for working around some of these inefficiencies.
- **Modular testing and development:** Each separate module can now be tested independently. Python also provides a method by which a module can detect if it is running as part of a larger project or is standing alone. With this method, each module can provide basic functionality when executed by itself to allow for testing of its isolated usability.
- **Efficient compilation:** Python uses pre-compiled object files (referred to as .Pyc files), which remove the need to translate the plain text files into the tokenised byte code. These optimisations, along with an optimised interpreter system, ensure that execution time is instantaneous. Unlike Perl, it has one method for compiling (following the Python method of there being 'one way to do it') that is optimised for its purpose.
- **Platform optimisation:** While the Python module system is not as advanced as the Perl CPAN network, Python does provide some methods of optimising its execution on other platforms. An example module is Psycho, which can optimise the execution speed on Windows platforms by 30%.

The Python prototype is the most stable and advanced of all the distributions and represents the final prototype against which the workspace transference framework can be evaluated.

Chapter 8. Conclusion

8.1 Thesis summary

The number of PCs we use on a daily basis is increasing, as is the need to manage these separate workspaces. A number of solutions can, with some effort, counter this problem. The creation of a home network for personal file sharing or the adoption of a virtualisation or roaming profile system within the workplace are possible ways to combat some of these issues. Unfortunately, existing solutions suffer a number of drawbacks, including the requirement of physical connection to a fixed network, and any disconnection effectively isolates the machine in question from the rest of the group.

The concept of workspace transference enables the user to transport his or her personal workspace across multiple machines regardless of physical proximity, local network restrictions and OS or hardware differences. Instead of the user having to replicate the workspace environment on each new machine, workspace transference provides a unified method to create and maintain one workspace. This workspace is effectively moved with the user rather than requiring installation of the user's preferred applications, settings and other configurations on every machine. These workspaces can be located on machines that are already owned and maintained by the user—a personal desktop computer or laptop, for example—or be within machines the user operates regularly such as a workplace PC.

As well as transporting the workspace between these machines, the applications contained within the workspace should be capable of adapting to relevant user-specified rules. Some settings, such as bookmarks and browser preferences, are common to all machines, while others should be selectively applied (e.g. a rule for personal pictures to be retained on home desktop machines only).

There is no currently existing solution to the problem of the transportation of applications and files across any number of machines regardless of hardware or software factors. In order to address this gap, this thesis has examined existing research and created a design to address these deficiencies. This design led to the construction of the workspace transference framework that states the requirements of such a solution.

As stated in the introduction, the original research question was:

How can one best create an operational workspace transference environment that is free of platform limitations and restrictions to support a typical suite of programs of any origin and benefit most users?

To answer this question, the constructed workspace transference framework was used along with the prototyping research methodology to gradually refine a series of prototype solutions. During this evolutionary process, each prototype implementation was evaluated against both a series of subjective persona tasks as well as more objective technical criteria.

To assess the viability of the prototypes, a number of personas were created in order to measure the prototype's functionality. Each of these personas, which represented a cross-section of the user base, described a number of use case tasks to be performed against the prototypes with the goal of uncovering any potential implementation issues. Using this method, the revised prototype, known in its implementation stage as WorkTran, was measured against these stringent requirements. Technical tests were also conducted to measure the programming language suitability, cross-platform programming and operation, GUI compatibility and other technical factors. As the prototypes evolved though the implementation cycles, an increasing number of these issues were resolved, leading to the revised prototype. Additional technical criteria were also employed to refine the prototype from the more objective standpoint of the underlying functionality not normally exposed to the testing process.

The prototype demonstrated a clear and practical purpose for real-world problems. During the use case walkthroughs, the revised prototype demonstrated the portability of many applications across OSs and on different hardware configurations. The successful workspace transference of the example applications such as the Firefox web browser and the OpenOffice word processor demonstrate that even these complex applications can be used within a portable workspace.

The revised prototype demonstrated that the workspace transference framework is not just an idealistic set of requirements but that its goals are achievable using current technology. The success of this implementation demonstrates the practicality of the workspace transference framework as a basis for a defined and provable standard of application and data portability.

8.2 Future research

The framework proposed in this thesis achieved its objective of providing a demonstrable prototype that meets the requirements stated in Chapter 3. The creation of the workspace transference framework and the construction of the prototypes used in this research can also provide a foundation for further work.

Several directions for extending and improving the models and implementation are suggested:

- **Extended cross-platform language research:** While Perl and Python were selected for the prototype languages, further research could be conducted to extend the investigation of cross-platform coding to other environments with the development of further comparable prototypes. The recent popularity of similar languages to Python and Perl including Java, Ruby, TCL and PHP would all make valuable studies of how these languages compare for providing portability solutions. These languages were not originally explored during the prototyping stage due to their elimination during the technology selection stage (see 6.1).
- **Package manager improvements:** The constructed prototypes implement an internal package manager, but the use of more established solutions such as Apt[159], YUM[160] or Fink[161] would greatly expand the implemented solution's capabilities. Incorporation of these project-extending abilities into the prototypes was considered during the earlier planning stages of this research, but the complexity of this implementation and the maintenance of the code base were thought to quickly overwhelm the project. The cross-platform conversion of an existing and proven solution such as Apt would be widely

considered a huge advantage to software distribution and management, but the added layer of complexity for different OSs (instead of its native Debian Linux) would present a problem for the initial implementers.

- **Decentralised distribution:** The reasoning behind the decision to use the client–server method of data distribution in the prototype design stage is to form a simple central point to deal with the complex issue of conflicts. These conflicts occur when two clients simultaneously update the same data block (e.g. a file). In such cases, the server would attempt to merge the two conflicting entities and distribute the corrected data. If a conflict cannot be resolved, the server attempts to retain the conflicting data in a way that keeps both clients operating but alerts the user to the potential problem. While adequate as a proof-of-concept, use of client–server architecture opens the implementation to issues such as bottle necks, networking latency or network disruption as well as the requirement of connectivity to the server. These issues could be addressed with the implementation of a peer-to-peer distribution system, which would allow individual peers to provide the server functionality.
- **Web-based platforms:** One emphasis throughout this research has been on the maintenance of a workspace centred around the collection of data within typical files and folders. Newer technologies such as those used in Web 2.0-based websites promote a complete move to online services rather than the use of traditional applications. In Web 2.0-based systems, files, folders and other storage mechanisms would not be stored on the machine the user is operating on but rather on the application's own website and then accessed remotely. Examples of this concept include Google Apps[162] and Windows Live[163]. Both of these solutions provide users with the means to work on a document, spreadsheet, presentation or other such office document entirely online with no localised storage needed. Further investigation into web-based systems, which was briefly covered in the literature review (see Chapter 2), would uncover possible further avenues of investigation with regard to portable workspaces.

Appendix

A Glossary of terms

The following list shows various acronyms and terms used throughout this paper.

Term	Definition
Atomicity	The concept that a collection of data objects should be treated as one distinct unit. When applied to *packages* this concept refers to the entire contents of a package being available at once rather than as each individual component becomes available.
Bootstrap	The initial startup process of software when launched. Used within the WorkTran prototype to provide a small program (usually the language object code executer) to start and pass full control to the cross platform compatible object code.
Candidate solution	A potencial piece of software compatible with the workspace transference framework.
CPAN	Comprehensive Perl Archive Network.
CPU	Central Processing Unit.
DB	Database.
Efficacy	The subjective self-measure of computer competency. Efficacy is drawn from a number of measures including outside encouragement, support, anxiety and outcome expectations. It is used in this research to ranking generated personas against with their own subjective competency.
ELF	Executable and Linkable Format. The standard executable format used within UNIX systems. Contrasted with EXE, the Microsoft Windows format.
Emulation	Running an application in a self-contained environment such as a virtual machine.
Engine	Computing term for a middleware layer between the core concept of an application and the communication with the underlying hardware.
EXE	Short for Executable, this is the standard file extension on Microsoft Windows platforms to denote a compiled binary executable file. Contrasted with ELF, the UNIX format.
Expect	An executable for POSIX based systems which interacts with 'dumb' terminal programs to simulate user input.
FOSS	Free and Open Source Software.
Framework	Used within the thesis to denote the proposed solution to a gap in research. This research proposes one such framework, namely the Workspace Transference Framework, for use in solving the research question discussed in the Introduction.
FTP	File Transfer Protocol. A relatively old file transfer standard between a single client and server setup.
GNU	GNU is Not UNIX.
HTTP	HyperText Transfer Protocol.
Host	Used within this research to refer to the physical computing device the user is currently working on as their active workspace. For example a laptop computer.
Hot Desking	A computer user who does not require a specific and dedicated host. Instead any machine may be selected to work from.
IAM	Immediate Access Memory. Exceptionally fast RAM equivalent used for pipelining program execution instructions at the CPU level.
IDE	Integrated Development Environment.

Initial prototype	The first generation of prototype software created to demonstrate the workspace transference framework. The initial prototype was created within the Perl development envrionment, the details of which are discussed in Chapter 6.
IPC	Inter-process Communication. The process whereby one application can communicate with a separate application.
Multiplexer	A programming technique where a piece of software acts differently depending on environmental settings. In the context of the prototype, the term refers to a library being used for each operating system in order to wrap OS level functionality regardless of which host OS is active. This presents a consistent API across multiple platforms.
NC	Network Computer.
Netbook	A smaller and more economical laptop computer aimed specifically at mobility and portability.
OS	Operating System.
Package	Conceptual term used within the prototype to define a collection of data and/or applications that confirms to the workspace transference framework.
Pass-though	In contrast to emulation, when an application is already suitable for the local physical hosts environment a 'null' like emulation can occur where instructions from the executing application can be run directly within the compatible host OS.
PC	Personal Computer.
PDA	Personal Digital Assistant.
Persona	A representative of a given user population[88]. See 4.1 for a full definition.
Podding	Moving an application from one machine to another by moving its executable and data.
Portability	The ease of which a program can be transferred from one environment to another[8].
POSIX	Portable Operating System Interface [for UNIX].
Predict / PyPredict	A specially created, cross platform equivalent of the Expect POSIX system. See 6.3.1 for further details.
RAM	Random Access Memory
RDiff	RSync Difference. A computed low-bandwidth patch between two files across a network.
Revised prototype	The second generation of prototype software created to demonstrate the workspace transference prototype. The revised prototype was created within the Python development envrionment, the details of which are discussed in Chapter 7.
RSync	Popular tried-and-tested GNU method of file synchronisation.
SSH	Secure Shell.
SSL	Secure Socket Layer.
Stasis	Freezing a program during run-time, maintaining its RAM contents.
Teleportation	Keeping the executable of an application in one location but moving the output GUI to another machine.
U3	A White paper for application podding[16] and the physical hardware simulating both a standard read/write USB disk and read-only faux CD-ROM.
UML	User-Mode Linux.
USB	Universal Serial Bus.
VM	Virtual Machine.
VM Farm	Virtual Machine Farm. A collection of larger servers sharing a number of VM's as well as a common data source and networking. The arrangement allows optimised VM migration for load balancing between each machine within the farm or to prevent a single host hardware failure disrupting the contained VM's.

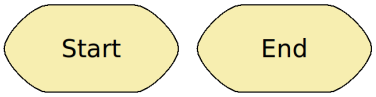
Web proxy	A server which sits between the requesting HTTP client and destination server. The web proxies purpose is to relay requests between the two and provide a caching system in order to minimize the number of outgoing duplicate requests.
WINE	Wine Is Not an Emulator / Windows Emulator
Workspace Transference	An extension of *Portability* whereby any data or application can be moved between multiple machines used by a computer operator. The potential workspace environment can be contained on any machine regardless of whether it is personally owned, a company asset or a temporary or one-off use machine.
WorkTran	The revised prototype detailed in Chapter 7.

B Logic Model

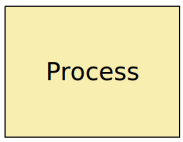
The prototype logic model is expanded from the prototype design (see Chapter 5) and discussed in greater detail using both a series of conceptual logic flow diagrams as well as a breakdown of each logical stage.

These logical planning diagrams represent the internal logic of each of the constructed prototypes constructed planned during the design chapter and constructed in the prototype implementation chapters (see Chapter 6 & Chapter 7).

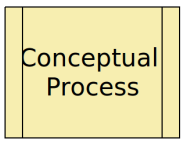
B.1 Key and definitions



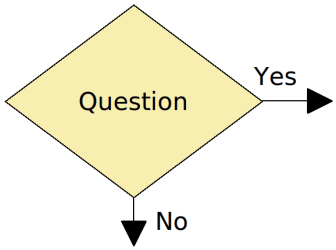
Determines the logical start or end of the process.



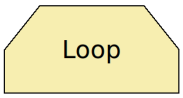
Describes a computational process within the system.



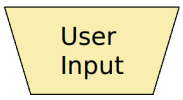
Similar to the above process symbol but indicates that the inner workings are specified on a sperate logic diagram.



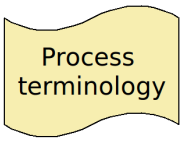
Shows a logical decision within the logic tree. Each branch is represented with a suitable answer and logical flow indicator.



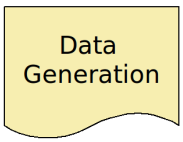
The multiple operation indicator implies that the next logical stage will be repeated based on the condition specified within the symbol.
An escape test may also be specified in the form of the above Question logic symbol.



User input indicator. Implies that this logical stage cannot continue until a user input is received.



Displays a conceptual process related to the overall system operation.
This typically indicates a thread fork where the subsequent operations occur asyncronously.



A process that has mass data generation as its output.

Figure 23. Symbols used in this chapter to denote the logical flow of execution

B.2 Overall system model

The logic diagram Figure 24 shows the main execution of the prototype from a macro level. Each major operation such as GUI, Watcher and Synchroniser are all discussed in their own respective logical models.

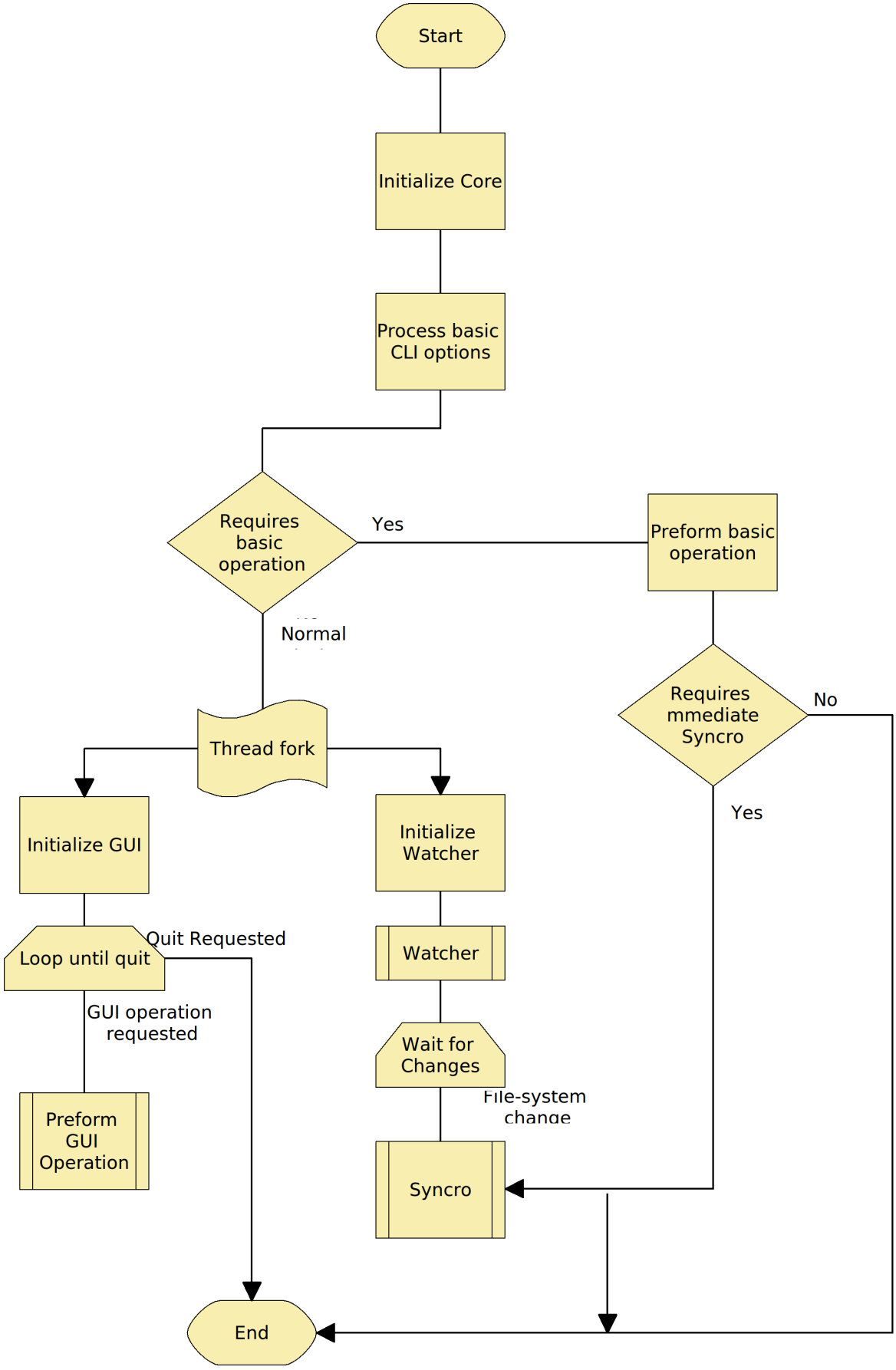


Figure 24. Overall conceptual logical flow diagram of the core system

B.2.1 Initialize core

This stage starts the main program, initializes variables, reads options and prepares the main system for execution. Global objects are setup at this point to be later shared with lesser modules. In particular the Options object which carries the basic program settings is initialized which loads the config XML file, checks the operating system and loads the appropriate wrapper libraries for non cross-platform operations such as file path styles and OS specific libraries.

B.2.2 Process basic command line options

The prototype must be capable of answering basic questions as well as operating with stripped-down functionality. In this stage various debugging settings may be specified such as how detailed logging should be or which UI system should be loaded.

At present the two major prototype user interfaces are Wx (for a graphicaly-window-based interface) and CLI (for a console-based interface).

This ties in with the basic operating stage below as a means of simply instructing the prototype to perform a benign task to which the calling processes does not need a great deal of feedback beyond regular console output.

B.2.3 Requires basic operation

Should the program be executed with a simple task (e.g. 'sync once') or is the normal interface required? This is determined alongside the previous stage as a method for single operations which can be scripted.

This functionality is provided as an easier way of instructing the prototype to perform a relatively benign task without having to engage in a higher level communication system in order to perform it.

Traditionally a local API system would be constructed (which the prototypes both provide) in order to perform bi-directional communication between the prototype and any third-party component wishing to use its services. However for simple operations, execution from the command line is also permitted.

B.2.4 Perform basic operation

As a continuation of the previous stage, this internally performs the action requested, if the program is running as a stand-alone. If an existing normal-mode version of the prototype is currently running - the command is passed up to this version and the command line version silently yields control to this parent.

Essentially this informs the already active prototype component to perform a previously set method of operation. This is the equivalent of the calling process initializing a COM / ActiveX / API Object and calling the method manually. These functions are performed internally which allows a relatively easy method to script the prototype.

B.2.5 Requires immediate synchronisation

Questions whether the previous operation requires a synchronised operation to be performed before returning execution control back to the parent process. Some operations - such as installing, upgrading or simply a manually invoked synchronisation - all require a full synchronisation cycle to be marked as complete. This stage determines whether control should pause until that operation completes or whether the program should quit immediately. This is largely a logic-based determination with the incoming instruction and whether that instruction had any actual effect on the portable environment.

For example the command 'install package X' would obviously require a synchronisation with the server in order to actually pull package 'X' onto the local client machines. However if 'X' is already installed this is obviously not necessary.

B.2.6 Thread fork

A conceptual entity to denote simultaneous execution of a GUI and the background Watcher processes.

In most operating systems the GUI sub-system is handled at an operating system level but requires a frequent yield (Python/Perl terminology), DoEvents (Visual Basic) or some other method which informs the OS that the GUI event queue should be checked for user input.

This is largely a remnant of earlier GUI-based operating systems where a single operating system thread or process would be expected to maintain its various screen elements (drawing the relevant items, responding to user interaction and operating system or other application events) as well as performing its actual mandate. Modern GUI-based OS's, generally separate the drawing and maintenance tasks out to an OS level handler - buttons are all drawn alike after all so these should be handled by one process rather than by each application having to implement its own helper. Windows Vista / 7 and recent GNOME and KDE releases have all taken this approach but for sake of compatibility a simple GUI handler has to be maintained to handle user and OS level input.

B.2.7 Initialize user interface

The GUI system is prepared for use. With the Wx GUI interface this involves loading the core Wx libraries, checking integrity and pre-allocating memory to future objects.

Since the Wx system is already loaded in the prototype binary, since this is not an especially time consuming activity, this can be seen as a period where the application can initialize and prepare its own objects for display as well as taking care of some of the more arduous one-such as the decompression and allocation of picture data etc.

Even relatively benign items such as the application icon (generally shown as the icon in the top left of each window) has to be saved in a universal format, which with Wx is usually binary data. This obviously has to be converted into whatever format the active operating system prefers - earlier Windows systems preferred a simple palette coded bitmap file, whereas modern versions of

Windows as well as Linux-based systems such as GNOME or KDE all prefer alpha blended PNG files.

B.2.8 Loop until quit

Executes in a loop until the quit event is triggered by either the user or a system event.

This is realistically the main loop of the program, it is intended to remain dormant until acutely requested to do so, usually by the setting of a flag when the user interacts with a screen element.

B.2.9 Perform interface action

See the GUI logic model (section 1.2.3) for further breakdown of this logic block.

The majority of the code in this section is intended for Wx and the local OS GUI library would initially handle the control event (such as the animation of 'pressing down' a button) followed by the firing of the associated code-block bound to that item.

B.2.10 Initialize Watcher

The watcher is preloaded and prepared for action.

What the process involves, depends on the actual operating system in use. Within Windows environments, calls to the NotifyEvent API are made; on Linux INotify Kernel, calls are dispatched. Either method relies on a callback system from the operating system on the host machine.

B.2.11 Watcher

See the Watcher logic model sub-chapter for further breakdown of this logic block.

B.2.12 Wait for changes

Indicates a constant loop until the Watcher picks up a system level change which should be dispatched to the active Synchroniser module.

Like the above main loop, this function performs few actions other than the polling of the local operating systems file monitoring library. This differs based on which system the prototype is running on and could take the form of either an active poll-based system (earlier Windows versions) or a passive callback system (later Windows systems and Linux-based systems using iNotify).

B.2.13 Synchroniser

See the Synchronser logic model for further breakdown of this logic block.

B.3 Graphical user interface

This second level logic diagram shown in Figure 25 outlines the operation of the GUI sub-system modules. This input system is one of the two main UI base's methods (the other being command line scriptable input) used to control the program.

Since WX is used as the GUI system, some elements of this tree are maintained by the separate Wx project and do not form part of the main prototype core system.

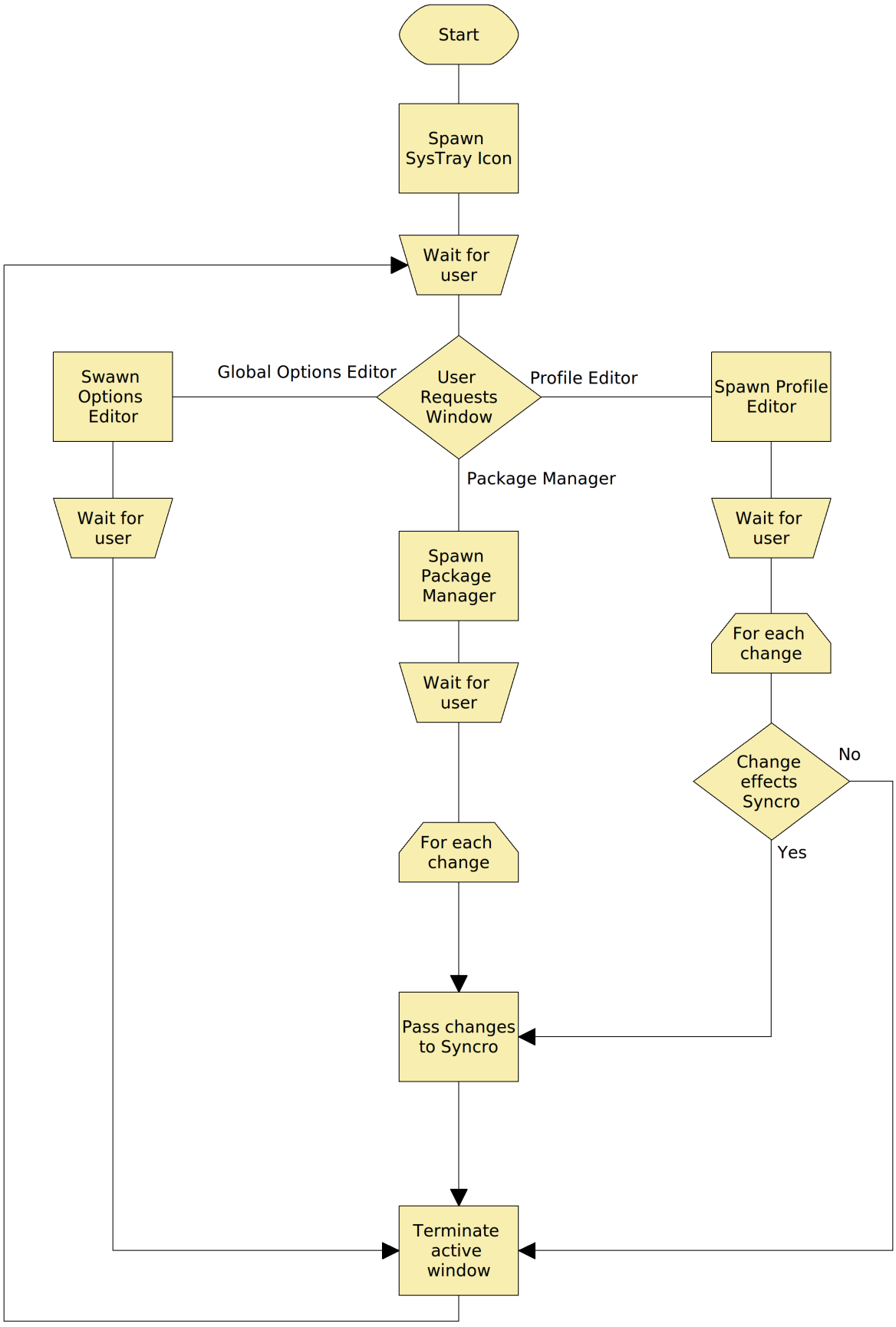


Figure 25. Logical flow diagram of the GUI sub-system

B.3.1 Spawn SysTray icon

Simple process to invoke the central, unobtrusive GUI point of presence for the user to communicate with the prototype.

This involves invoking the relevant Wx objects and attaching callbacks to the various handles provided by the Wx object hierarchy.

B.3.2 Wait for user

After the above SysTray icon has been spawned, the program simply waits in the background while the Watcher and Synchroniser processes act on any outstanding operation requests. From the user's perspective, further GUI stages are only invoked upon user request via the SysTray.

B.3.3 User Requests Window

The user selected an action that requires I/O from a GUI window. At this point we can spawn any windowing interface supported by the GUI sub-system.

This logical block is actually a dispatcher which prepares a common GUI template window and then hands control over to the specified window decorator class responsible for its actions.

B.3.4 Spawn Options Editor

Displays the global options window. Since all changes in this window are immediate the Synchroniser does not need to be informed. Instead the changes will become active on the Synchroniser's next execution cycle.

B.3.5 Spawn Package Manager

This represents the central interface point for the user.

The package manager allows installing, removal and management of any portable software package within the portable environment. The central job of this windowing interface is to display a filterable list of packages and to wait for user input.

B.3.6 Wait for user

This stage simply waits for a GUI level events to be trigged by the user.

B.3.7 For each change

This stage, in either the package manager or profile editor, indicates that changes should be passed onto the Synchroniser sub-system should further action from that module be necessary. This usually occurs when a number of files need to be installed from the server or removed from the client as the package action dictates.

B.3.8 Pass changes to Synchroniser

This stage essentially filters any changes made and selectively either breaks a currently executing Synchroniser operation or re-starts with the new settings.

B.3.9 Spawn Profile Editor

This operation spawns the profile manager Wx interface. This windowing system allows the editing of profile level options such as ignoring file changes to certain environmental profiles.

B.3.10 Change effects Synchroniser

With the profile editor, changes may be made to other profiles which are not necessarily relevant to the existing active profile. This filter determines if the changes need to be passed to the current Synchroniser system immediately or simply bypassed.

B.3.11 Terminate active window

This stage destroys the window, cleans up the released memory and resumes normal background idle behaviour.

B.4 Watcher

The watcher process shown in Figure 26 is a sub-system of the prototype that analyses system changes, determines if they are relevant to the portable environment and acts on them accordingly. The system watcher is intended for Read-Only access to the existing system and simply informs one of any higher level process of any changes.

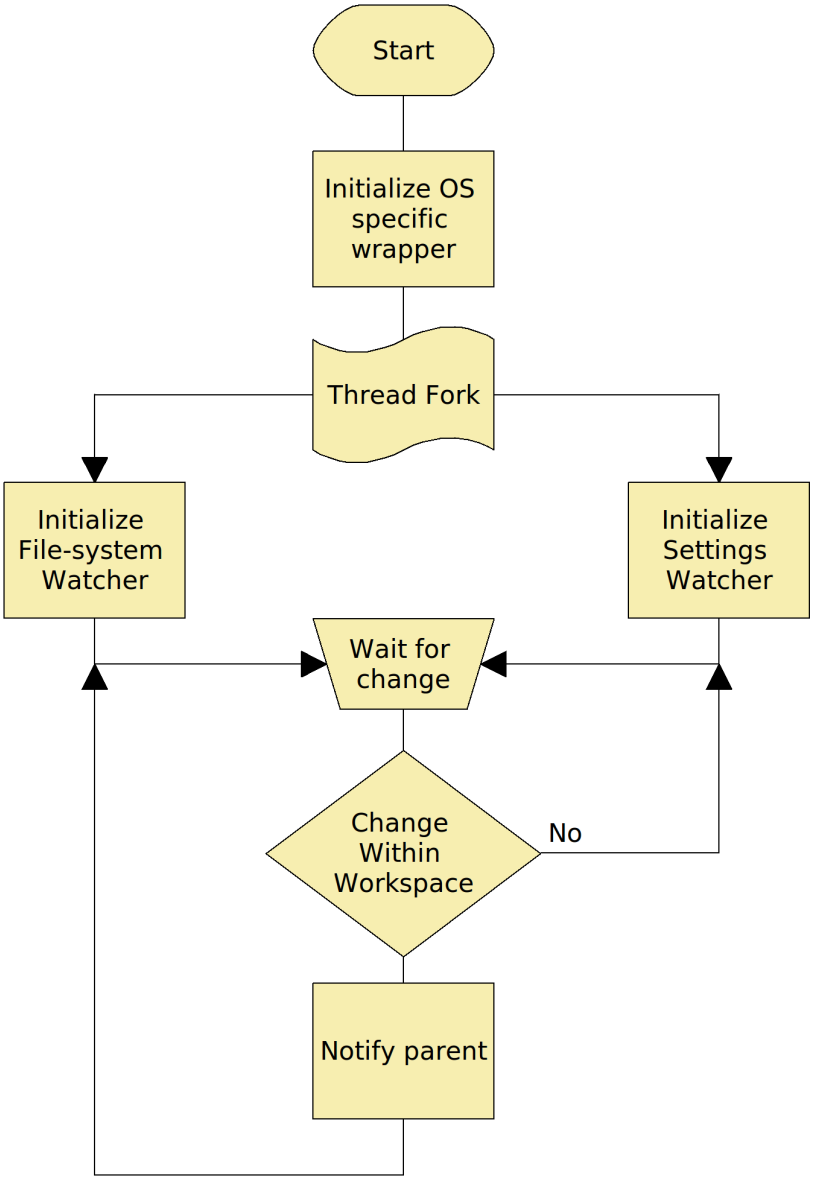


Figure 26. Logical flow diagram of the watcher sub-system

B.4.1 Initialize OS specific wrapper

The watcher is a standardized modular interface which is in turn constructed from specific wrapper stubs per active operating system. Each major system is supported and is expected to have a watcher wrapper which passes any detected changes to the main watcher process.

Since there is no standardized way of detecting system level changes, this is the only way that each respective OS can be tracked for file-level or settings-level alterations.

B.4.2 Thread fork

A conceptual entity to denote simultaneous execution of a the file-system and setting's watching threads.

B.4.3 Initialize file-system Watcher

This process detects any file system changes within the environment. Any changes such as time-stamp changes, updates, creations or deletions are detected and passed on to any interested parent.

B.4.4 Initialize Settings Watcher

This process is only really relevant on programs that store information in un-controllable files such as the registry on Windows systems or within GEdit on Linux systems.

This watcher intercepts these changes and notifies any interested Watcher parents.

B.4.5 Wait for change

This process is individual to each file-system or setting's watcher thread. Each may execute a wait operation in its own way based on the active OS.

B.4.6 Change within workspace

Since in many cases some operations may be falsely detected by either of the two main Watcher threads this stage is an extra safeguard to filter out non-prototype environmental information before it is passed to the parent.

B.4.7 Notify Parent

Should any of the above threads detect a change and if it satisfies the conditions within the above decision, this process passes the detected change on to the Watcher parent. The parent, which is usually the Synchroniser system, then takes appropriate action to safeguard the data in a transferable way.

An example would be a registry change on a windows computer which would then, via the Watcher, be detected and stored in a more portable format to be used within another environment.

B.5 Synchronisation Manager

This Synchroniser system shown in Figure 27 represents the sub-system responsible for the saving and safeguarding of information within the workspace. The Synchroniser system ensures that information in one-location perfectly matches that of another.

This could be a client-server model or it could ensure that the temporary storage on the local machine matches a remote storage device such as a Bluetooth storage device or other 'slow' storage such as networked or VPN storage concepts.

Should a remote-to-local change be requested (e.g. upgrading a piece of in-use software) the upgradable components are downloaded into temporary storage (usually a hidden directory with the suffix '-new') and the changes that had been made when the files were open, are finally closed.

An example of this would be a queued upgrade of a package, where a contained .exe file is currently in use. This software could be in use by the user which would make it un-upgradable since most

OS's lock running program files. In this case, the differences between the current installation and the remote installation are analysed and the changes are downloaded into its hidden temporary directory. As soon as the Watcher sub-system detects that no contained file is open, the changes are made onto the system, upgrading it to the latest package version.

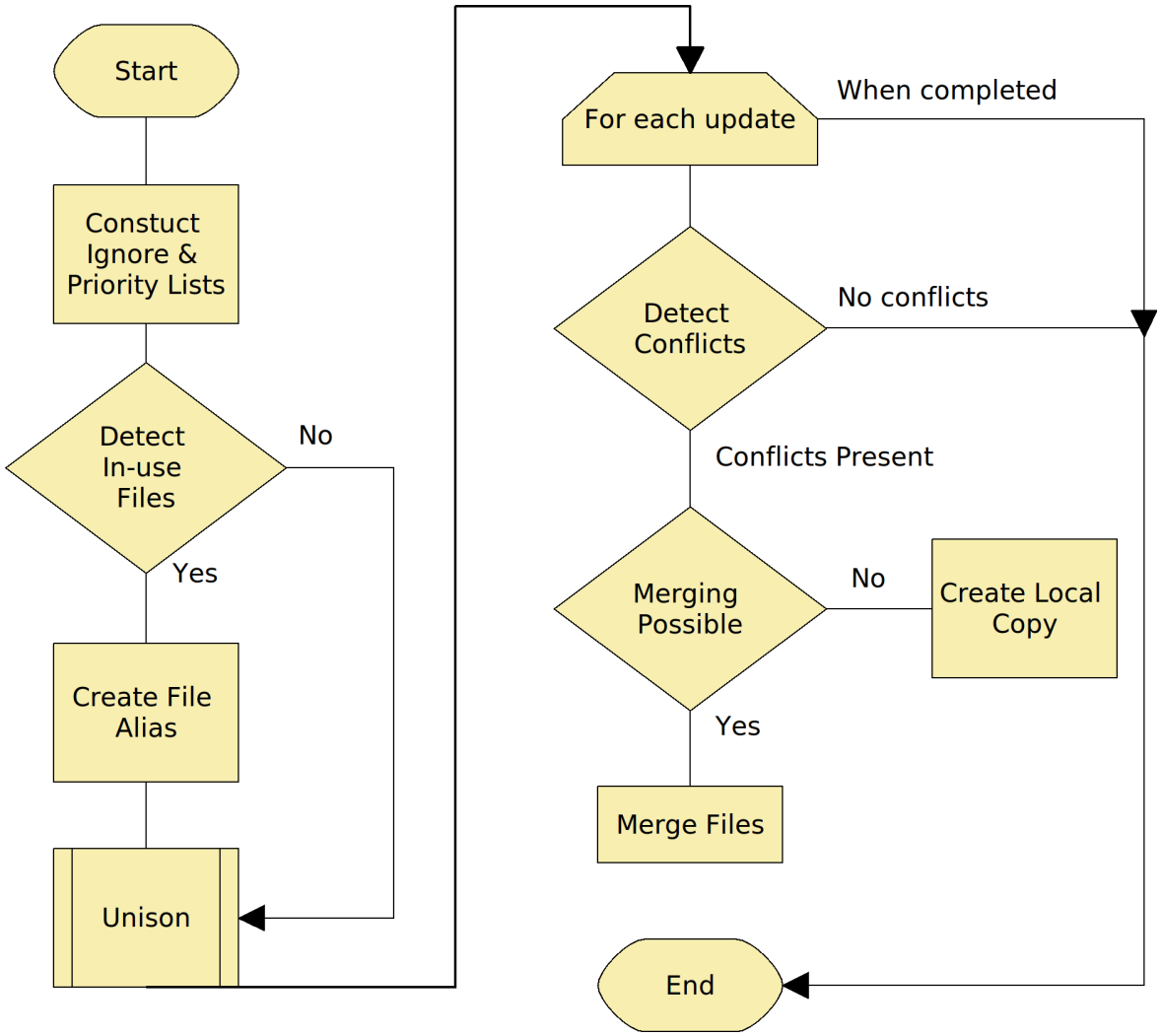


Figure 27. Logical flow diagram of the synchroniser sub-system

B.5.1 Construct ignore and priority lists

Each profile may have a list of priorities and ignores, based on the current rule set. In this stage those lists are parsed either from general rules (e.g. 'Ignore all changes to Spreadsheets') or specific file-system based rules (e.g. 'Documents are regularly changed so these have first priority').

B.5.2 Detect in-use files

Should any files be detected as being in-use (not just locked but open in any way) and a reverse Synchroniser action is queued (i.e. the server has a version that the client does not) a local copy of the information is created and the updates are queued in place.

Further information about this process is discussed in the main Synchroniser discussion at the beginning of the Synchroniser logic model (section 1.2.5).

B.5.3 Create file alias

As above, this stage refers to how the prototype handles in-use files when a remote-to-local upgrade occurs. See the main Synchroniser logic model (section 1.2.5) description for further information on this process.

B.5.4 Unison

When all Synchroniser operations are queued the Unison[71] process is executed which connects the local and remote processes and performs the actual RSync optimized transfer.

See the later Unison logic breakdown for detailed information on how the Unison/RSync programs resolved optimized difference hashes.

B.5.5 For each update

After the Unison operation has been concluded, each change needs to be analysed. In the case of upgrading software (see above) a patch needs to be conducted in order to merge the current and new packages together.

B.5.6 Detect conflicts

During the upgrading process, some files may not merge perfectly together. This usually occurs when setting's files (e.g. .ini or .cfg files) conflict between the packaged, factory default versions and the users own personalized installation.

B.5.7 Merging possible

During the merging process some files may not be capable of being merged in place. This often occurs when files conflict to a large degree in unmergeable formats such as non plain-text files.

B.5.8 Create local copy

The server version adopts the true file name with the local version being copied elsewhere or renamed should the user wish to retain it.

B.5.9 Merge files

For some file formats it is possible to resolve the differences using a simple patching system. If this is possible those patches are applied here.

Merging simple .ini or .cfg files is relatively straightforward and these files can be combined at this stage to support both past settings and new package option sets.

B.6 Unison

This section analyses how the Unison system works together with RSync in order to optimize differential hashes and to synchronise two file-systems in an efficient manner. Further information is available in the paper on RSync by Andrew Tridgell[68]. Figure 28 displays the overview of this sub-system.

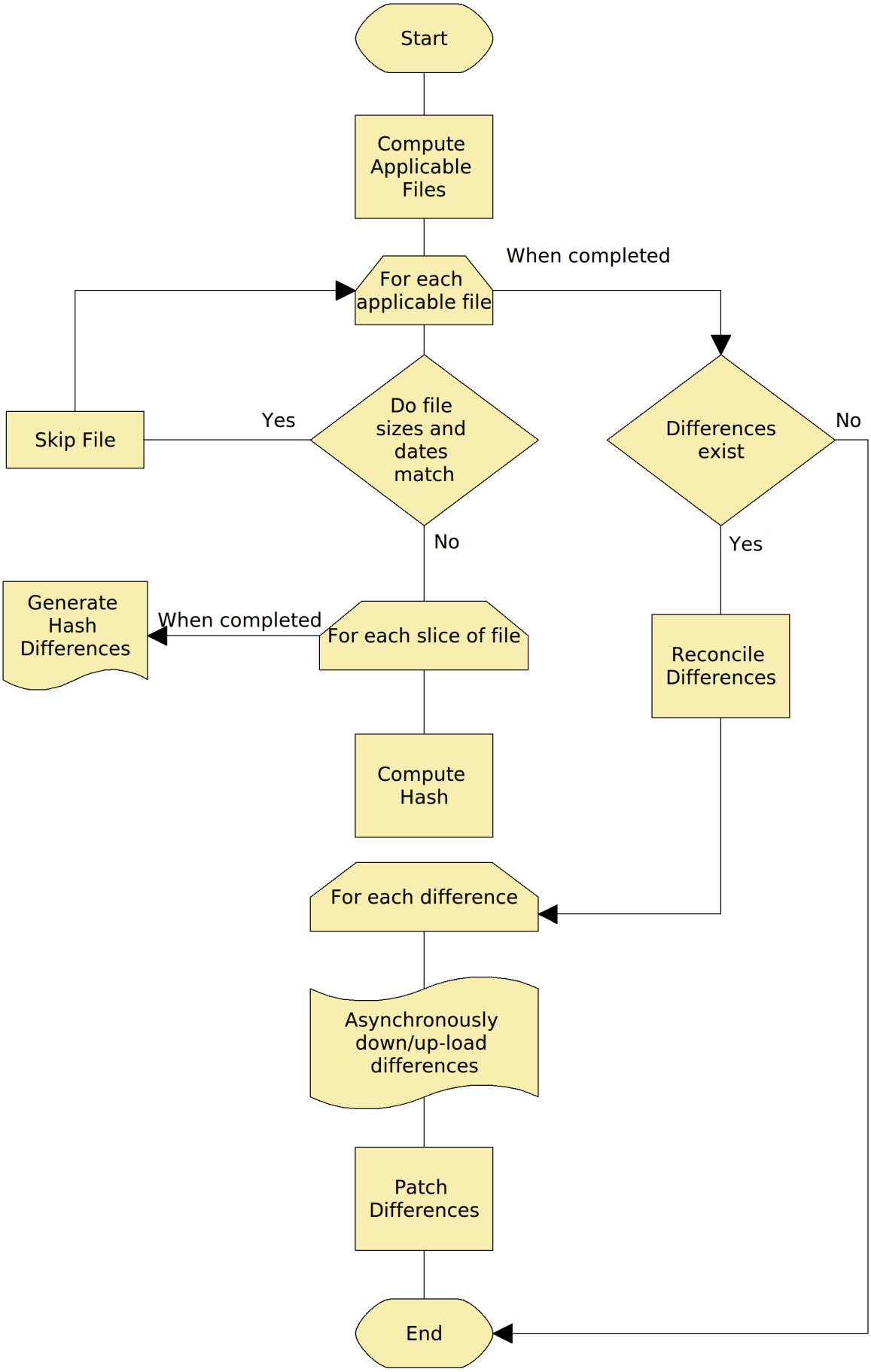


Figure 28. Logical flow diagram of the Unison sub-system

B.6.1 Compute applicable files

During this stage, files comparable on both the local and remote systems are listed. Any mismatches represent simple whole-transfer files which need to be copied to the relevant location on either the client or server.

It would theoretically be possible to implement a system whereby a file that was simply moved could be detected thus preventing the differential detecting the absence in one location and the creation of a 'new' file in the second. Ext2 and other file systems provide a simple unique identifier (in this case an integer) which never actually changes, providing the file allocation remains on the same storage device[164]. This can be used as an effective primary-key of the file and would provide some mechanism for detecting moved files. An unfortunate outcome of this process is that the file systems in widest use, such as the Microsoft FAT model, do not provide such a facility and make the whole process exceptionally cumbersome. Nonetheless this should be considered as a possible future avenue of exploration.

B.6.2 For each applicable file

For each file computed in the above stage the subsequent logical stages are performed.

B.6.3 Do file sizes and dates match

A simple indicator to detect at a glance if some of the files have changed. To adapt to this both the prototype and Unison executables are aware of common exceptions such as Excel Spreadsheets (XLS) and Access Databases (MDB) which can remain both the same size and omit to change the files modified date stamp. As a guideline though this rule works to quickly determine which files need to be examined in closer detail.

An additional problem is that on some operating systems (specifically any Microsoft Windows system) time is not completely accurate. Therefore 'fuzzy' analysis is required where time needs to be allowed to drift by an acceptable range to counteract the less precise time stamps.

B.6.4 Skip file

Should the file-sizes and modified date stamps match and the file is *not* an excluded file type (Excel Spreadsheets etc.), no further analysis is needed.

B.6.5 For each slice of file

Should a file merit further analysis, the file is then split into various slices and (as in the next stage) a hash is generated for each slice.

B.6.6 Compute hash

Each slice of the file has both a weak and strong MD4 hash in order to compute the client and server differences.

B.6.7 Generate hash differences

The result of this process is the creation of two comparable hash tables one on the client-side and one on the server-side. Both of these hash tables can then be analysed in the next stage.

B.6.8 Differences exist

As the *Do file sizes and dates match* stage (see 1.2.6.3) indicates, not all files will actually have been changed in any plausible way beyond the date-stamp changing. In this case files with client and server identical hash tables can be discarded.

B.6.9 Reconcile differences

Should differences occur this stage attempts by using the weak and strong MD4 hashes generated earlier, to compute exactly which slices of the two potentially similar files need to be sliced or spliced from the two copies.

B.6.10 For each difference

After the change table has been computed a number of download requests are queued in order to reconstruct the differences.

This conflict resolution is performed within the server and can differ by data types. For example a plain text file is relatively easy to reconcile but a binary file usually requires an external plugin to merge correctly.

An example of this are the older Microsoft Office format files (pre 2007 shift to DocX format) which generally require a more sophisticated approach to reconciliation where each file is read in and the files spliced with the aid of the office revision mechanism.

B.6.11 Asynchronously download/upload differences

In this stage the above changes are either downloaded from the changed remote server or uploaded from the newer client. The process used is largely that of the popular RSync algorithm[68] and differs only in that the above stage attempts to eliminate the need for this more expensive side-by-side comparison by using the size and date of the file as a guideline.

Briefly the RSync algorithm minimizes the transfer between the client and server by reading comparative file on both hosts. Reading each file and computing both a strong and weak rolling checksum. When the blocks are assumed to match, no action is performed. When mismatched, the algorithm attempts to determine if the data block is missing on the client or server; when this occurs, the extra data is compressed and transmitted to the party lacking the data. Repeating this process with the added use of a compression compliant connection (zlib is often used as part of the standard SSH connection) ensures that data is efficiently patched on either client or server machines.

B.6.12 Patch differences

All changes are now present on both the client and server files. Both file images may now be patched to represent one-another precisely. This is accomplished by creating a new file in a temporary

location and merging the destination and patch files. Finally, the destination file is overwritten, creating a resultant merge between the two sources.

The upside of this process is that a sudden loss of power or other file-system level interruption (e.g. removal of the storage medium before writing has finished) is compensated for since this the process is atomic. The writing either finishes his project and the new file overwrites the old one or the temporary file is left in place and is then cleaned up on the next iteration.

The downside is the obvious extra use of diskspace, especially with large files and writing time on slower devices. The required temporary space is equal to the size of the previous version of the file (as it is not being changed until it is replaced) as well as the new patched file. A rough size estimate of 2x the original size of the file should be assumed for block level files, with the worst case scenario being potentially unlimited. Write speeds are not always consistent across devices, with newer solid state media such as SSD cards[165] having a comparatively long write time compared to the quicker read time. An optimized system could possibly provide a fragmentation based system which would optimize the write time for these devices, although at the expense of file recovery in the event of a system or disk crash.

C Scoping Prototype

This section details the creation of an initial scoping prototype created in the earlier stages of the design and implementation stages. This scoping prototype was created after the prototype design in Chapter 5 but before the creation of the initial prototype in Chapter 6.

Visual Basic was used for the scoping prototypes construction as it is the language that is the most familiar to the author and represents a lesser learning curve for rapid deployment of an initially testable prototype. This initial prototype was developed to investigate the boundaries of the project by defining what areas need to be covered in later design and implementation stages, and what would need to be examined in more depth in future prototypes. Early scoping of the prototype in this way also assisted in the construction of an appropriate timetable for this research project.

In this section the implementation and evaluation details of the scoping prototype will be explained, starting with the specific particulars of the prototypes development environment: Visual Basic 6.0. After the creation of the prototype the developed software will then be tested against the tests generated in the evaluation criteria chapter (see Chapter 4).

Due to the purely speculative nature of this prototype the persona use case tasks were omitted from the testing stages in favor of the much more mature initial prototype (see Chapter 6). This section instead reviews only the more technical aspects of constructing a prototype based on the workspace transference framework.

C.1 Module implementation

This section discusses the scoping prototype's implementation stage from the design in Chapter 5. These list the experiences during the construction phase using the Visual Basic programming environment.

C.1.1 Constants and options

The Visual Basic language does not provide any suitably portable XML reader. In order to process XML Visual Basic relies on the Microsoft XML library which is the standard shared library intended for use by all system components when processing XML. This library provides the usual COM interface[166] compatible with most languages which can communicate according to the COM protocol and is seen as the Microsoft version of the SAX[167] and Expat[168] (see the later Perl based prototype in Chapter 6) open-source equivalents. Unfortunately this library; depending on operating system version, service pack and whether or not Microsoft Office is installed, may or may not be available within the client environment. Some of the relevant library hooks also differ from one version to another so the expected behaviour can radically differ when reading the XML file data on these multiple platforms.

This limitation is by no-means a fault of the Visual Basic development platform itself but can instead be attributed to the strategy Microsoft employs in distributing 'shared' components within its operating system[166]. Microsoft has also stated that this model is subject to change as the operating system is updated with backwards compatibility damaging existing libraries[169].

In light of these facts and also taking into consideration the relatively simple nature of the XML format, it was significantly easier to simply develop a light-weight Visual Basic XML reader which could nativity process the configuration files internally. This is obviously not an optimal solution, but did provide the portability, speed and ease of use needed to work with these simple files without depending on the dubious COM component.

C.1.2 Fundamentals and cross-platform support

The Visual Basic implementation, as befits a Microsoft language, is limited to the Microsoft Windows operating system. Recent efforts however, such as WINE[46] (confusingly: "WINE Is Not an Emulator") has enabled 'pure' Windows executables such as Visual Basic compiled EXE files to run under other environments such as Linux or Mac operating systems via a POSIX[170] interface. Consequently, the prototype can safely assume the Windows file storage methodology with all other translations being performed by the emulation layer (in this case WINE).

WINE provides a translation between the native Microsoft Windows object code and the operating systems POSIX (Universal API) layer. This translation is not 100% compatible so WINE must attempt to provide the comparable Windows C API calls that are expected by the executing program. Unfortunately, due to the age and evolution of some of these older APIs, these can be inconsistent or produce side-effects which WINE must also replicate.

Due to these considerations WINE is still in a process of development, trying to completely emulate the Windows environment. However later versions have demonstrated clear success for most programs.

This has both good and bad aspects. The positive side to this implementation is that no platform specific coding need take place when WINE automatically ensures that any operating system specific functionality is made to work under whatever 'real' operating system the host is running. The downside is that while this emulation ensures that the prototype works well with all Microsoft Windows operating systems, this also means that no support whatsoever is provided for non-Windows platforms, except though any exposure that WINE provides. This can vary wildly depending on distribution and is not guaranteed in later WINE releases due to security concerns or local WINE configurations.

To extend the analogy of differences in file systems from 5.1.3, WINE works around the Windows drive letter allocation by creating a fake 'C' drive which resembles a normal Windows installation. This was obviously created for the purpose of operating system stability with common directories such as 'Program Files' and the 'Windows' directory being located where a normal program would assume to find them. A link to the real host operating system file tree is (again this can depend on WINE distribution) provided via a special Z drive which maps directly onto the root of the main host file system. In these cases the location of the *true* document storage path is dependent on a number of issues with WINE:

- The compiled version of WINE supporting the this setup
- WINE correctly mapping the documents folder on the fake drive C to the local file tree OR
- WINE enabling the Z drive
- WINE allowing access to the outer operating systems file system for writing via its virtual drive Z

With the above requirements and the implementation of various security limitations in the newer releases of the WINE platform, it is not a complete guarantee that any access is given to the true file storage directory.

In these cases the scoping prototype can only be judged a success on Windows platforms but a dubious one on any other.

C.1.3 Synchronisation

The Visual Basic project attempted to be as portable as possible with an implementation of the synchronisation module by implementing such as system in the Visual Basic environment. This proved problematical due to the complexity of the synchronisation scheme, LibRSync simply does not support non-POSIX environments and cannot be ported to Visual Basic. Due to this a reinvention of RSync had to be undertaken with the VB language. This was time consuming to develop and is obviously less efficient than the compiled C implementation.

While moderately successful in its implementation the creation of an equivalent RSync algorithm within the Visual Basic language suffered efficiency and speed issues unlike its original C++ library. The load on the CPU was unacceptable as was the differential computation time and RAM resources required in order to perform these scans. While beneficial to have the entire RSync algorithm managed internally, thus giving more control to the process, the number of workarounds and optimization techniques far outweighed the cost of implementing it within this prototype.

C.1.4 Networking functionality

Visual Basic 6 was quite late to the concept of Internet-based technologies with its previous 5 versions neglecting the concept entirely. Even the 6.0 release of the language offers very little in the way of core language support for Internet-based activities, instead relying on common operating system-based technologies such as WinSock[171] which provided an object orientation-based event interface for socket programming

While socket communication did provide the language with a certain level of support for client-server communications, WinSock's complexity was a hindrance to smaller projects and frequently meant that developers seeking to utilize the its functionality were forced to either code a wrapper library (which occurred in the case of the scoping prototype) or rely on a commercial equivalent. WinSock's inconsistent distribution and buggy implementation in some of the earlier versions of the Windows XP system caused frequent problems and sometimes required quite complex workarounds. In one notable case Windows 98, Windows NT4 and Windows XP (until service pack 2) all had incorrect WinSock licensing information and required the installation of a Microsoft supplied patch in order to function correctly[172]. This patch requires administrative level privileges and thus was out of the reach of most university students and office workers who could not install the patch due to lack of the requisite privileges.

As an alternative to this potential dead-end, attempts were made to address the requirement of a full installation, the most popular being the CSocketClass[173] which, instead of using the faulty ActiveX WinSock control, directly used the low-level Windows API in order to create and manipulate the operating system socket library.

In the case of the scoping prototype, the CSocketClass was used to provide networking functionality but its continuing buggy nature meant that some workarounds had to be discovered and implemented at great cost to development time.

C.1.5 GUI Development

Visual Basic was one of the first languages to embrace the concept of event-driven programming. As such the Visual Basic implementation was quite simplistic and relatively efficient, the language being specifically built around the GUI interface[174]. This was greatly aided by the strong emphasis on GUI-led design providing both a graphical IDE for the design of all screens within the project as well as a coding system strongly tied to the front end development platform.

A minor point of the GUI development in Visual Basic however is the exact positioning of elements on the screen. While suitable for a fixed resolution this strategy can quickly become a hindrance, especially on devices which can only support lower horizontal or vertical screen resolutions such as netbooks. The latter prototypes combat this problem by allowing a flexible GUI system which flows around the available screen area rather than using the exact positioning.

C.1.6 Technical criteria summary

The Visual Basic version of the prototype can be considered a success but the lack of networking functionality compared with the other two higher level languages is a severe hurdle to viability outside of simple prototyping. Its lack of threading support renders it slow and unwieldy during large operations and its instability on non-Microsoft systems hampers its long term practicality.

The Visual Basic implementation was suitable as a simple test to see what was possible but is not practical for the larger prototype implementation.

C.2 Technical criteria

This section discusses the technical criteria generated during the evaluation criteria chapter (see Chapter 4).

C.2.1 Distribution

Since the Visual Basic language was designed to function exclusively on the Windows platform with a VB5 or VB6 library, the executables produced by the Visual Basic compiler, are small and relatively compact. Using the 'EXE' format (short for *executable*) large projects can be efficiently transported as a single file.

Unfortunately since Visual Basic is limited to a single operating system, it is not transferable to any other environment without the assistance of an emulator such as WINE.

Like most Windows executables all resources (graphics, controls and language settings) can be compiled into a single EXE file which can easily be transported between machines. The additional step of compressing the executable further with a system like UPX[150] can provide further space savings at a slight cost to boot speed.

Another minor point with Windows executable files is that the binary file itself can contain additional information such as the author, copyright information and an icon. This allows the operating system to prompt using this data when administrative privileges are required. An example of this would be when installing to a drive or directory space not writable by a normal non-administrative user. In this case the OS will display a message asking the user if they are sure they wish to install the program using the executable's embedded authorship information.

The Visual Basic libraries (approximately 1mb for Visual Basic 5 and 1.5mb for VB6) are provided as standard on all Microsoft platforms. No installation sequence is necessary for the core Visual Basic language.

Unfortunately Visual Basic uses early binding of external resources such as DLL files which make the use of external controls problematic without bundling the language with an installer. Such installers automatically register external DLL's and correctly install the Visual Basic program, however this also presents a number of obstacles:

- **The Installer is un-customizable** - No additional questions can be asked of the user except basic installation questions such as installation directory (usually somewhere under the 'Program Files' directory tree).
- **The Installer is designed for larger projects** - The installer itself is primarily designed for larger projects rather than the projects being designed to be space saving or portable. While not a major problem of itself, this can mean that regular installer metrics such as progress bars (which fill as quickly as they appear on the screen for smaller applications) can largely be a waste.
- **Designed for single-install projects** - There is no means to install on removable media or to allow the installer to provide extra features for removable disks. Install directories are always assumed to be within the regular 'Program Files' path.
- **Always assumes administrator level** - All the Visual Basic installers assume that the application installer must have Administrator privileges. No provision for installing in the user's own personal storage space is made. If the user does not already possess administrator privileges then the installer will prompt for the login credentials of a user who does. While a minor point in itself with personally administered machines, this can represent a significant problem for users in workplaces, cyber cafes or other situations where the user privileges are restricted for security or policy reasons.

Due to these considerations the Visual Basic executable is not, in its initial form, truly mobile and represents a portability problem for non-administrator users trying to use the prototype in its un-installed form.

The failure of the Visual Basic distribution is an intrinsic restriction of the Visual Basic language itself rather than being a fault of the prototype, it must be noted that should the user already have administrator access most of the above enumerated problems do not occur.

C.2.2 Synchronisation

The mechanism whereby files can be uploaded via a HTTP interface to a remote server was largely successful for both small and extremely large files.

However the upload mechanism is incredibly inefficient within the Visual Basic language and some of the inherent bugs within the process can be of serious detriment to the overall concept of portability for this prototype. This stems from an issue with the WinSock (Windows Socket library) which does not work correctly with any Visual Basic application that calls upon it unless a patch from Microsoft is applied first. This process requires administrator privileges which, as already discussed, are not always a safe assumption.

Likewise, downloading large binary portions of data is relatively unstable using the WinSock libraries provided to Visual Basic.

This problem seems to originate from the poor handling of data within the Visual Basic language and from the poor implementation of memory-to-disk data flushing. Incoming binary data is often mangled and unrecognizable, even when various encoding standards were enforced such as ISO 8859-1 or UTF8.

In order to work around the problems with raw binary data, Base64 encoding was applied to prevent this data corruption. Base64[175] is a method whereby binary data is encoded to use only the 24 letters of the alphabet plus numerical digits (0-9 as characters) and four 'safe' punctuation characters. Using this standard, data can safely be moved from one computer to another in a 'sanitized' form and the binary data can be completely recreated at each end of the transmission. While Base64 encoding presents an overhead it is a standard for any binary-unsafe operation. Email attachments work with Base64 for example, and provide a mechanism to effectively work around any binary encoding problems that may be present on the source, destination or at any platform in between. This encoding standard, while foolproof, is not meant for large volumes of data, since it presents a 33% increase in data size.

These methods were employed as workarounds to the larger problem of dealing with incoming data sockets in Visual Basic \ WinSock and while successful they represent a large networking and functional overhead which is unacceptable as a means of data transfer.

C.2.3 Package management

Visual Basic is by its nature a visual, event-based language and GUI interfaces presented little or no issues when constructed. The largely drag-and-drop interface presented by the programming languages IDE system allowed for the creation of the GUI system demonstrates the VB6 environments strength.

The package installation sequence relies strongly on the regular data download mechanism (discussed above) and thus exhibited the same problems with large quantities of binary data. The additional overhead of a 33% increase of data (due to the Base64 encoding standard) added to the file-sizes of the incoming data but ultimately presented no real concerns in the final use.

The single operating system restriction for Visual Basic made it relatively easy to aim the prototype and the single operating system platform supported making the whole package installation sequence uncomplicated as no provisions had to be made for other operating systems including their pathing systems or API-based quirks.

Even though the download of the data stores presented major issues for efficiency, the package installation sequence itself was successful within the scoping prototype.

The package removal process generally involves the recursion into that packages directory structure with the removal of each file and folder encountered. Finally any references the package may have left behind are removed. These can include locally stored registry keys or prototype level configuration (which files the package should register itself with for example).

Largely an extension of the download process, automatic upgrading of packages was ultimately successful in that data could be transferred, albeit with some trade-offs in efficiency.

Since the upgrading procedure involves generating a differential over the local file store followed by the same process on the server, this process was again hampered by the Base64 decoding process which made the storage mechanism largely incompatible with server side software - in this case the RSync daemon used to generate these differentials. There was another workaround for these issues but this again represents a needless layering of software to fix a fundamental problem on the Windows platform.

After these workarounds were researched and implemented the automatic upgrading mechanism worked successfully and with a degree of efficiency although the increased overhead on CPU and RAM resources on the client and server were far greater than they needed to be.

C.2.4 Error handling

The WinSock libraries and their integration with Visual Basic were originally designed in the earlier days of the Internet and thus lack some of the more robust handling for unexpected events. WinSock deals with disconnections using the same method as a network timeout making it exceptionally difficult to detect if a connection fault has occurred with the connection to the server or whether the packet simply needs re-sending. Only by manually testing the connection on a subsequent connect event is it possible to determine the fault's cause.

This extra stage represents a workaround to a problem that could easily be solved by the upstream WinSock libraries but the acceptance and widespread workarounds for this deficiency will most likely prevent a fix as other applications already compensate in this manner.

The extra code and research time involved in finding solutions for these minor problems is largely unnecessary and represents a serious deficiency with both the Visual Basic language, the WinSock libraries and the Microsoft Windows operating system to which WinSock belongs.

Although these workarounds must be present to detect network disconnection this does not present any major problem with network disconnection handling. Additional checks must be made by the prototype to ensure that the network is still active rather than timing out but these extra stages present few problems for the process of detecting such an event.

All executables within Windows are loaded into RAM on execution so the removal of the underlying storage media has no effect on the program. Additionally since the Visual Basic language is Windows centric, calls to the Windows API layer can detect when the underlying device has been removed and efficiently manage the graceful close of open resources on the withdrawn media and therefore prevent data loss.

The removal of the underlying storage device presents no major issue to the language itself. Additional information (such as the device being ejected, by whom and why) are available through the Windows API which can provide these as well as other details. This enables the prototype to

act accordingly. One minor consideration to this is that while packages containing other programs can receive the shutdown signal from the prototype it is entirely up to that program as to whether or not to act upon it. Due to this, some packaged applications may present their own problems if the underlying media is removed.

Aside from these minor issues, the prototype successfully detects media disconnection correctly and efficiently.

C.2.5 Development environment

The Visual Basic environment is intended as a rapid development and deployment environment and demonstrates these goals quite easily. While not forthcoming in the number of examples or tutorials available, its relatively clean syntax and graphically-based programming environment make for a rapid development speed.

In particular the Visual Basic GUI system represents one of the cleanest and easily learned functions of a programming language available today. Unfortunately the language is marred by some basic omissions that have been available to other languages for a comparatively long time. Lack of established base data types such as hashes and arrays can quickly cause problems and involve the continuous reinventing of functionality that should be provided within the language itself. The language is also slow and inefficient, largely because of its semi-interpreted nature and its over-reliance on COM+ and poor operating system interfaces. This in itself does not present any problem to the experienced programmer who has learned the ways around these problems but getting to that stage can take time and a great deal of experience.

While Visual Basic does follow an Object Orientation philosophy the actual implementation of this is quite loose to say the least. Modules are difficult to maintain and the lack of any true encapsulation, abstraction or introspection make the language quite dated by the standards of other development environments. Over-reliance on proprietary black-box external elements such as COM+ objects also place severe constrains on debugging an active program. This is shown mainly by some of the Windows networking COM+ objects which are hampered by the Windows operating system itself which sets a poor example in regard to a clean and concise API standard.

Some errors which these protocols are years old and still exist in even modern operating systems such as Windows 7. API's are often poorly documented with the assumption that the programmer referencing them has insider knowledge into the Windows operating system, a luxury not afforded to third party developers. Likewise some of these API's provide no further updates from their first invention and provide no means to comment or clarify within normal channels.

The overall impression is of a language that is easy enough to learn but anything more complex than the basic application can require detailed Windows API workarounds to fix the neglected deficiencies in the language itself.

Despite these deficiencies, as a programming language Visual Basic does provide a decent implementation and rapid development speed.

C.3 Prototype evaluation & weaknesses

The first tentative prototype was based on a simple conceptual model built in the language most familiar to the researcher, Visual Basic. This conceptual model attempted to implement the workspace transference framework's synchronisation guidelines into a workable demo.

While suitable as an early conceptual implementation this prototype was intended as a simple conceptual test to quickly outline both the scope of the specification of the research and to analyze what tools exist to implement the workspace transference framework.

The Visual Basic implementation was reasonably efficient but strongly linked to its native platform - Microsoft Windows. In order to extend the solution to another OS it becomes necessary to implement a WINE emulation layer[46] on top of the program binary to enable a fully cross platform system. This has a number of disadvantages, not least of which is the fact that the Visual Basic core system comprises of a large (by program executable standards at 1.5mb) external dynamic link library (DLL) which must be present and registered on the host system. The actual 'compiled' executable is simply a object-code-based binary[176] which calls the main DLL to actually execute out the instructions contained in the program. This intercommunication is cumbersome even on its native platform, giving the Visual Basic language a reputation for its relative slowness and inefficiency against other higher level languages.

Within an environment such as WINE, this process suffers further though additional impedance of the cross-platform emulation adding increased overhead to the CPU and RAM causing the program to run more slowly than it would on its native system. Interestingly recent optimizations with the WINE system have led certain executables (in specific scenarios) actually running faster and more efficiently on Linux-based systems than they otherwise would on their native Windows-based systems [47].

Excepting the problems with speed and cross-platform deficiencies the scoping prototype was an rudimentary method of establishing the scope of the project but not viable for a completed demonstration of the workspace transference framework.

The conclusions drawn from its implementation are as follows:

- **Cross Platform Issues** - While the native Windows API provides a large quantity of operating system specific functionality some functions do not cross operating system boundaries. Device dependent calls such as releasing hardware from the system queue (similar to 'ejecting' a USB device from the system by using the Remove Device icon on Windows systems) rarely survive the platform barrier as WINE is unable to map them to the POSIX[170] equivalent.
- **Networking Issues** - Visual Basic features extremely limited networking abilities which are prone to data corruption during large data transfers. Issues such as incorrect parity calculations, data truncations or packet loss are inexcusable in a modern language and expose unnecessary frustration during simple network intercommunication. The additional insistence of the Visual Basic language of having the communication libraries outside the core system, places an additional strain on distribution with the requirement that these too must also be shipped with the executable.

- **Reinventing established concepts** - Server communication protocols such as RSync[68] provide no usable method of interaction with Visual Basic and must be reimplemented in the language. Coupled with the networking issues discussed above, this becomes troublesome to implement and maintain.
- **Quick and Efficient GUI Design** - A strength of the scoping prototype, using the Visual Basic IDE[177], provides efficient GUI design.
- **Relatively trouble free implementation** - While only relevant on Windows systems, the compiling of Visual Basic executable is relatively simple and efficient. An executable compiled in one flavour of Microsoft operating system is more-or-less guaranteed to run correctly in another. In regard to cross-platform compatibility, with the assistance of WINE, the executables are also fairly reliable (albeit relatively slow and inefficient in some cases, see above). The compatibility is fairly transparent with the possible exception is any specific low-level API call to the Windows system, some of which are not natively mapped by WINE in the current releases. Polling executing processes and checking for the presence of physical hardware represented the two areas that WINE is somewhat lacking from its emulated environment.
- **Well documented** - Due to its maturity, the Visual Basic language is well documented and supported both by Microsoft and by third party developers who also provide tutorials and example code. The GUI system, using the standard Microsoft Forms library[178] follows well-defined standards and has numerous usage examples.
- **Portability** - Executables, assuming the presence of the Visual Basic core DLL files and other external interfaces, work correctly and reasonably in most systems. Unfortunately some third-party modules (including the external Microsoft communication libraries used for client-server communication) insist on full administrative-level system installation before being functional. In some circumstances this can be established without administrator access but the complexity in installing these modules is slow and fault-prone as well as the functionality sometimes being restricted by system policy. Since the assumption is that all applications go through an installation sequence before being able to run, there is sometimes (again depending on policy) no effective way of bypassing this unfortunately necessary stage. This decision is due to the belief that during the platforms inception, all programs would be distributed via a standard installation kit rather than simply being distributed by the required executable. This attitude is now changing and smaller more specific, single use programs are now regaining popularity. While Microsoft are coming around to this mindset, Visual Basic 6 was abandoned for the .NET suite and thus remains behind with these new advances.
- **Poor Error Handling** - Due to the maturity of the Visual Basic language, its error handling capabilities were included late in the programming languages life, leading to poor implementation and an all-or-nothing approach to error resolution. Error handling in Visual Basic is essentially limited to a very simplistic implementation of the 'On Error Goto' methodology which provides no structure for cleaning up after an error other than what the coder implements. Errors not 'caught' or expected are fatal to the program's execution and result in immediate program termination[179].

Again, the Visual Basic implementation is intended as a simple trial environment to analyze the scope of the project and not intended to represent the final prototype. Nevertheless, the above conclusions are relevant to all implementations and future prototype success can be judged on the comparative success in eliminating these troublesome issues.

The most apparent conclusion from the evaluation of the scoping prototype is the use of the Visual Basic language itself. Visual Basic is efficient in some areas such as prototype design, language documentation and GUI implementation. But its lack of support for networking, its COM system (the modules requiring a full-scale installation to become active) and its requirement on external modules which are not always reliable, position it as a language best left to the standard installation model of application distribution and not suitable as a portable environment.

In addition to the language limitations it also becomes apparent that integrated implementation of the synchronisation protocol, RSync, represents the majority of the complexity and bug tracking within the completed prototype. Exporting this complexity to a separate module or executable maintained as part of the main RSync or Unison[71] projects would be preferable to taking the full implementation into the prototype.

These issues aside, the above does not imply that the Visual Basic language is totally unsuitable. Its GUI centric design and simple object orientation model signifies that an prototype can be constructed in minutes compared to even the simplest C, Python, Perl, Java or any other higher level language equivalent. The lessons learned from this implementation have been invaluable and in other circumstances the Visual Basic platform would form a suitable basis if it was not for the very specific requirements that this project demands.

From the above we can conclude that the prototype implementation is to be repeated with the following amendments:

- The chosen programming language is to be one with a better understanding of network communication, ideally this should be integrated into the core.
- The chosen programming language is to have the lowest possible number of external modules in order to reduce reliance on modules which may fail.
- A more efficient environment is required that does not rely on an emulation layer to operate (in this case WINE).
- An environment is required where cross-platform coding is provided as much as possible.
- An environment is required where error catching is more efficiently implemented and where smaller errors are non-fatal to the continuing execution of the prototype.

References

1. Computing in the clouds, Weiss, Aaron p. 16-25, 2007, New York, NY, USA, http://doi.acm.org/10.1145/1327512.1327513
2. Remote Desktop Protocol (RDP) Features and Performance, Microsoft Corporation, 2007, http://www.microsoft.com/technet/prodtechnol/Win2KTS/evaluate/featfunc/rdpfperf.mspx
3. Perspective layered visualization of collaborative workspaces, Shiozawa, Hidekazu, Okada, Ken-ichi and Matsushita, Yutaka, *GROUP '99: Proceedings of the international ACM SIGGROUP conference on Supporting group work (1-58113-065-1)* p. 71-80, ACM, 1999, Phoenix, Arizona, United States, New York, NY, USA, http://doi.acm.org/10.1145/320297.320305
4. Network Computers-Ubiquitous computing or dumb multimedia?, Herrtwich, R. G. and Kappner, T., *ISADS '97: Proceedings of the 3rd International Symposium on Autonomous Decentralized Systems (0-8186-7783-X)* p. 155, 1997, IEEE Computer Society, Washington, DC, USA
5. PortableApps.com, Rare Ideas LLC, 2007, http://portableapps.com
6. MojoPac, RingCube Technologies Inc., 2007, https://www.mojopac.com
7. Computer use and ownership - CPS Reports, Education & Social Stratification Branch, 2008, http://www.census.gov/population/www/socdemo/computer.html
8. Portability and adaptability, Poole, P. C. and Waite, William M., *Software Engineering, An Advanced Course (3-540-07168-7)* p. 183-278, 1975, Springer-Verlag, London, UK
9. Evolution in the design of abstract machines for software portability, Thalmann, Daniel, *ICSE '78: Proceedings of the 3rd international conference on Software engineering* p. 333-341, 1978, IEEE Press, Atlanta, Georgia, United States, Piscataway, NJ, USA
10. Information for students: key events in Microsoft history, Microsoft Visitor Center Student Information, 2005, http://www.microsoft.com/about/companyinformation/visitorcenter/students.aspx
11. Experience with porting the Portable C Compiler, Minchew, Charles H. and Tai, Kuo-Chung, *ACM 82: Proceedings of the ACM '82 conference (0-89791-085-0)* p. 52-63, 1982, ACM Press, New York, NY, USA, http://doi.acm.org/10.1145/800174.809758
12. P-STAT overview, P-STAT Inc., 2009, http://www.pstat.com/lookat.html
13. P-code and compiler portability: experience with a Modula-2 optimizing compiler, Louden, Kenneth p. 53-59, 1990, ACM, New York, NY, USA, http://doi.acm.org/10.1145/382080.382632
14. Portable Firefox, John Haller, 2007, http://portableapps.com/apps/internet/firefox_portable
15. PortaPuTTy, Bryan L. Fordham, 2007, Richmond Hill, Ga., http://socialistsushi.com/portaputty
16. U3 deployment kit, U3 LLC, 2007, Redwood City CA, USA, https://www.u3.com
17. U3 deployment kit, U3 LLC, 2006, Redwood City CA, USA, https://www.u3.com/developers/technical/default.aspx
18. WebPod: persistent Web browsing sessions with pocketable storage devices, Shaya Potter and Jason Nieh, *WWW '05: Proceedings of the 14th international conference on World Wide Web (1-59593-046-9)* p. 603-612, ACM Press, 2005, Chiba, Japan, New York, NY, USA, http://doi.acm.org/10.1145/1060745.1060833
19. The design and implementation of Zap: a system for migrating computing environments, Steven Osman, Dinesh Subhraveti, Gong Su and Jason Nieh p. 361-376, 2002, ACM Press, New York, NY, USA, http://doi.acm.org/10.1145/844128.844162
20. The V distributed system, David Cheriton p. 314-333, 1988, New York, NY, USA, http://doi.acm.org/10.1145/42392.42400
21. Reincarnating PCs with portable SoulPads, Ramon Caceres, Casey Carter, Chandra Narayanaswami and Mandayam Raghunath, *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services (1-931971-31-5)* p. 65-78, 2005, ACM Press, Seattle, Washington, New York, NY, USA, http://doi.acm.org/10.1145/1067170.1067179
22. Internet Suspend/Resume, Kozuch, Michael and Satyanarayanan, M., *WMCSA '02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications (0-7695-1647-5)* p. 40, 2002, Washington, DC, USA
23. A break in the clouds: towards a cloud definition, Vaquero, Luis M., Rodero-Merino, Luis, Caceres, Juan and Lindner, Maik p. 50-55, 2009, ACM Press, New York, NY, USA, http://doi.acm.org/10.1145/1496091.1496100
24. VMware business infrastructure virtualization: beyond virtual machines & Servers, 2009, http://www.vmware.com/
25. VirtualBox, Sun Microsystems, 2009, http://www.virtualbox.org/
26. Windows VirtualPC, Microsoft, 2009, http://www.microsoft.com/windows/virtual-pc/default.aspx
27. Mastering Microsoft Virtualization, Tim Cerling and Jeffrey Buller, *9780470449585* p. 253-312, 2009, John Wiley and Sons
28. Controlling data in the cloud: outsourcing computation without outsourcing control, Chow, Richard, Golle, Philippe, Jakobsson, Markus, Shi, Elaine, Staddon, Jessica, Masuoka, Ryusuke and Molina, Jesus, *CCSW '09: Proceedings of the 2009 ACM workshop on Cloud computing security (978-1-60558-784-4)* p. 85-90, 2009, ACM, Chicago, Illinois, USA, New York, NY, USA, http://doi.acm.org/10.1145/1655008.1655020
29. VMware ACE - Assured Computing Environment, VMware, Inc., 2009, http://www.vmware.com/products/ace/
30. The Case for VM-Based Cloudlets in Mobile Computing, Mahadev Satyanarayanan, Paramvir Bahl, Ramón Cáceres and Nigel Davies p. 14-23, 2009, IEEE Pervasive Computing, Los Alamitos, CA, USA, http://doi.ieeecomputersociety.org/10.1109/MPRV.2009.82

31. GreenCloud: a new architecture for green data center, Liu, Liang, Wang, Hao, Liu, Xue, Jin, Xing, He, Wen Bo, Wang, Qing Bo and Chen, Ying, *ICAC-INDST '09: Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session (978-1-60558-612-0)* p. 29-38, 2009, ACM, Barcelona, Spain, New York, NY, USA, http://doi.acm.org/10.1145/1555312.1555319
32. Implementing and operating an internet scale distributed application using service oriented architecture principles and cloud computing infrastructure, Sedayao, Jeff, *iiWAS '08: Proceedings of the 10th International Conference on Information Integration and Web-based Applications \& Services (978-1-60558-349-5)* p. 417-421, 2008, ACM, Linz, Austria, New York, NY, USA, http://doi.acm.org/10.1145/1497308.1497384
33. Securing elastic applications on mobile devices for cloud computing, Zhang, Xinwen, Schiffman, Joshua, Gibbs, Simon, Kunjithapatham, Anugeetha and Jeong, Sangoh, *CCSW '09: Proceedings of the 2009 ACM workshop on Cloud computing security (978-1-60558-784-4)* p. 127-134, 2009, ACM, Chicago, Illinois, USA, New York, NY, USA, http://doi.acm.org/10.1145/1655008.1655026
34. Optimizing the migration of virtual computers, Sapuntzakis, Constantine P., Chandra, Ramesh, Pfaff, Ben, Chow, Jim, Lam, Monica S. and Rosenblum, Mendel, *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation* p. 377-390, 2002, ACM, Boston, Massachusetts, New York, NY, USA, http://doi.acm.org/10.1145/1060289.1060324
35. X Window System (panel session), James Gettys, Goerges Grinstein, Bertram Herzog and Robert Scheifler, *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques (0-89791-275-6)* p. 349, 1988, New York, NY, USA, http://doi.acm.org/10.1145/54852.378554
36. Teleporting in an X Window System Environment, T. Richardson, F. Bennett, G. Mapp and A. Hopper, *IEEE Personal Communications*, 1994
37. XMOVE: a pseudoserver for X window movement, Ethan Solomita, James Kempf and Dan Duchamp p. 143-170, 1994, O'Reilly & Associates, Sebastopol, CA, USA
38. Easy access to remote graphical UNIX applications for windows users, Repasky, Richard R., *SIGUCCS '04: Proceedings of the 32nd annual ACM SIGUCCS conference on User services (1-58113-869-5)* p. 357-359, 2004, Baltimore, MD, USA, New York, NY, USA, http://doi.acm.org/10.1145/1027802.1027886
39. Easy access to remote graphical UNIX applications for windows users, Richard R. Repasky, *SIGUCCS '04: Proceedings of the 32nd annual ACM SIGUCCS conference on User services (1-58113-869-5)* p. 357-359, 2004, Baltimore, MD, USA, New York, NY, USA, http://doi.acm.org/10.1145/1027802.1027886
40. XLiveCD, Dick Repasky, 2006, IN, USA, http://xlivecd.indiana.edu/
41. The interactive performance of SLIM: a stateless, thin-client architecture, Brian K. Schmidt, Monica S. Lam and J. Duane Northcutt, *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles (1-58113-140-2)* p. 32-47, 1999, ACM Press, Charleston, South Carolina, United States, New York, NY, USA, http://doi.acm.org/10.1145/319151.319154
42. An end-system approach to mobility management for 4G networks and its application to thin-client computing, Leo Patanapongpibul, Glenford Mapp and Andy Hopper p. 13-33, 2006, ACM Press, New York, NY, USA, http://doi.acm.org/10.1145/1148094.1148097
43. Limits of wide-area thin-client computing, Albert Lai and Jason Nieh, *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems (1-58113-531-9)* p. 228-239, 2002, ACM Press, Marina Del Rey, California, New York, NY, USA, http://doi.acm.org/10.1145/511334.511363
44. On the performance of wide-area thin-client computing, Albert M. Lai and Jason Nieh p. 175-209, 2006, ACM Press, New York, NY, USA, http://doi.acm.org/10.1145/1132026.1132029
45. MiniVMac, Paul C. Pratt, 2006, http://minivmac.sourceforge.net/
46. WINE, Bob Amstadt and Eric Youngdale, 2007, http://www.winehq.org
47. Wine Benchmarks 0.9.5, Tom Wickline, 2006, http://wiki.winehq.org/BenchMark-0.9.5
48. Running Windows viruses with Wine, Matt Moen, *NewsForge*, 2005, http://os.newsforge.com/article.pl?sid=05/01/25/1430222
49. Personal Information Everywhere (PIE), Boaz Carmeli, Benjamin Cohen and Alan J. Wecker, *HYPERTEXT '00: Proceedings of the eleventh ACM on Hypertext and hypermedia (1-58113-227-1)* p. 252-253, 2000, ACM Press, San Antonio, Texas, United States, New York, NY, USA, http://doi.acm.org/10.1145/336296.336502
50. A feather-weight virtual machine for windows applications, Yang Yu, Fanglu Guo, Susanta Nanda, Lap-chung Lam and Tzi-cker Chiueh, *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments (1-59593-332-6)* p. 24-34, 2006, ACM Press, Ottawa, Ontario, Canada, New York, NY, USA, http://doi.acm.org/10.1145/1134760.1134766
51. Xen and the art of virtualization, Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt and Andrew Warfield, *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles (1-58113-757-5)* p. 164-177, 2003, ACM Press, Bolton Landing, NY, USA, New York, NY, USA, http://doi.acm.org/10.1145/945445.945462
52. Diagnosing performance overheads in the xen virtual machine environment, Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman and Willy Zwaenepoel, *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments (1-59593-047-7)* p. 13-23, 2005, ACM Press, Chicago, IL, USA, New York, NY, USA, http://doi.acm.org/10.1145/1064979.1064984
53. Using coLinux to provide a linux environment on windows PC in public computer labs, Hideo Masuda, Michio Nakanishi, Akinori Saitoh and Seigo Yasutome, *SIGUCCS '06: Proceedings of the 34th annual ACM SIGUCCS conference on User services (1-59593-438-3)* p. 221-224, 2006, ACM Press, Edmonton, Alberta, Canada, New York, NY, USA, http://doi.acm.org/10.1145/1181216.1181266
54. Teaching operating systems administration with user mode linux, Renzo Davoli, *ITiCSE '04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*

(1-58113-836-9) p. 112-116, 2004, ACM Press, Leeds, United Kingdom, New York, NY, USA, http://doi.acm.org/10.1145/1007996.1008027
55. WormTerminator: an effective containment of unknown and polymorphic fast spreading worms, Songqing Chen, Xinyuan Wang, Lei Liu and Xinwen Zhang, *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems (1-59593-580-0)* p. 173-182, 2006, ACM Press, San Jose, California, USA, New York, NY, USA, http://doi.acm.org/10.1145/1185347.1185371
56. LiveCD List, FrozenTech, 2007, http://www.frozentech.com/content/livecd.php
57. LiveCD - Community Ubuntu Documentation, Canonical Ltd, 2009, https://help.ubuntu.com/community/LiveCD
58. FedoraLiveCD, Fedora Project, 2009, https://fedoraproject.org/wiki/FedoraLiveCD
59. Live CD - openSUSE, Novell Networks, 2009, http://en.opensuse.org/Live_CD
60. Operating systems on a stick, Shade, Eric p. 151-158, 2009, USA
61. The FreeBSD LiveCD Project, Brazilian FreeBSD User Group, 2009, http://livecd.sourceforge.net/
62. Market share, profit margin, and marketing efficiency of early movers, bricks and clicks, and specialists in e-commerce, Sungwook Min and Mary Wolfinbarger p. 1030-1039, 2005, http://www.sciencedirect.com/science/article/B6V7S-4CKNPJ9-1/2/eac2074597b85062ea6015b0c9c5cee5
63. Google Gears, 2009, http://gears.google.com/
64. HTML5, W3C, 2009, http://dev.w3.org/html5/spec/Overview.html
65. Endpoint security: managing USB-based removable devices with the advent of portable applications, Fabian, Michael, *InfoSecCD '07: Proceedings of the 4th annual conference on Information security curriculum development (978-1-59593-909-8)* p. 1-5, 2007, ACM Press, Kennesaw, Georgia, New York, NY, USA, http://doi.acm.org/10.1145/1409908.1409935
66. What is a file synchronizer?, S. Balasubramaniam and Benjamin C. Pierce, *MobiCom '98: Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking (1-58113-035-X)* p. 98-108, 1998, ACM Press, Dallas, Texas, United States, New York, NY, USA, http://doi.acm.org/10.1145/288235.288261
67. Optimism and consistency in partitioned distributed database systems, Susan B. Davidson p. 456-481, 1984, ACM Press, New York, NY, USA, http://doi.acm.org/10.1145/1270.1499
68. Efficient Algorithms for Sorting and Synchronization, Andrew Tridgell, 1999, ACM Press, http://samba.org/~tridge/phd_thesis.pdf
69. Diff, Patch, and Friends, Michael K. Johnson p. 2, 1996, Seattle, WA, USA
70. How RSync Works, Andrew Tridgell, 2000, http://samba.org/rsync/how-rsync-works.html
71. File synchronization with unison, Erik Inge Bolso p. 6, 2005, Seattle, WA, USA
72. ActiveSync - Windows Mobile Synchronization, Microsoft, 2007, http://www.microsoft.com/windowsmobile/activesync/default.mspx
73. Don't be lazy, be consistent: Postgres-R, A new way to implement database replication, Bettina Kemme and Gustavo Alonso, *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases (1-55860-715-3)* p. 134-143, 2000, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA
74. Synchronizing a database to improve freshness, Junghoo Cho and Hector Garcia-Molina, *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data (1-58113-217-4)* p. 117-128, 2000, ACM Press, Dallas, Texas, United States, New York, NY, USA, http://doi.acm.org/10.1145/342009.335391
75. MIDDLE-R: Consistent database replication at the middleware level, Marta Pati\&\#241;o-Martinez, Ricardo Jim\&\#233;nez-Peris, Bettina Kemme and Gustavo Alonso p. 375-423, 2005, ACM Press, New York, NY, USA, http://doi.acm.org/10.1145/1113574.1113576
76. Scalable synchronization of intermittently connected database clients, Wai Gen Yee and Ophir Frieder, *MDM '05: Proceedings of the 6th international conference on Mobile data management (1-59593-041-8)* p. 299-303, 2005, ACM Press, Ayia Napa, Cyprus, New York, NY, USA, http://doi.acm.org/10.1145/1071246.1071295
77. NTFS technical reference, Microsoft Corporation, 2009, http://technet.microsoft.com/en-us/library/cc758691%28WS.10%29.aspx
78. POLIPO: Policies & OntoLogies for Interoperability, Portability, and autOnomy, Daniel Trivellato, Fred Spiessens, Nicola Zannone and Sandro Etalle, *978-0-7695-3742-9* p. 110-113, 2009, ACM Press, Los Alamitos, CA, USA, http://doi.ieeecomputersociety.org/10.1109/POLICY.2009.19
79. Netchannel: a VMM-level mechanism for continuous, transparentdevice access during VM migration, Kumar, Sanjay and Schwan, Karsten, *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (978-1-59593-796-4)* p. 31-40, 2008, ACM Press, Seattle, WA, USA, New York, NY, USA, http://doi.acm.org/10.1145/1346256.1346261
80. wxWindows for cross-platform coding, Taran Rampersad p. 6, 2003, Seattle, WA, USA
81. Object-oriented programming in TCL/TK, Isaacson, Peter C. p. 206-215, 2001, USA
82. Qt GUI Toolkit: Porting graphics to multiple platforms using a GUI toolkit, Eng, Eirik p. 2, Seattle, WA, USA
83. Is a modern, robust Windows XP lab environment better than an older, simpler Windows NT4 environment?, Valenzuela, Bert, *SIGUCCS '02: Proceedings of the 30th annual ACM SIGUCCS conference on User services (1-58113-564-5)* p. 294-297, 2002, ACM Press, Providence, Rhode Island, USA, New York, NY, USA, http://doi.acm.org/10.1145/588646.588727
84. Specification and modeling: an academic perspective, Broy, Manfred, *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering (0-7695-1050-7)* p. 673-675, 2001, IEEE Computer Society, Toronto, Ontario, Canada, Washington, DC, USA
85. Should Computer Scientists Experiment More?, Walter F. Tichy p. 32-40, 1998, IEEE Computer Society Press, Los Alamitos, CA, USA, http://dx.doi.org/10.1109/2.675631

86. A design theory for systems that support emergent knowledge processes, M.L. Markus, A. Majchrzak and L Gasser p. 179-212, 2002, MIS Quarterly
87. The Inmates Are Running the Asylum: Why High Tech Products Drive Us Crazy and How to Restore the Sanity (2nd Edition), Alan Cooper, *0672326140*, 2004, Pearson Higher Education
88. Personas: practice and theory, John Pruitt and Jonathan Grudin, *DUX '03: Proceedings of the 2003 conference on Designing for user experiences (1-58113-728-1)* p. 1-15, 2003, Management Information Systems Research Center, San Francisco, California, New York, NY, USA, http://doi.acm.org/10.1145/997078.997089
89. Persona based rapid usability kick-off, Khalayli, Nina, Nyhus, Silja, Hamnes, Kari and Terum, Tone, *CHI '07: CHI '07 extended abstracts on Human factors in computing systems (978-1-59593-642-4)* p. 1771-1776, 2007, ACM Press, San Jose, CA, USA, New York, NY, USA, http://doi.acm.org/10.1145/1240866.1240898
90. Personas and efficacy, Deborah R. Compeau and Christopher A. Higgins p. 189-211, 1995
91. The impact of users' cognitive style on their navigational behaviors in web searching, Khamsum Kinley and Dian W. Tjondronegoro, *15th Australasian Document Computing Symposium (ADCS)* p. 68-75, 2010, School of Computer Science and IT, RMIT University, University of Melbourne, Melbourne, Victoria
92. Concept of the Corporation, Peter F. Drucker, *1560006250* p. xvii, 1983, Transaction Publishers
93. GZIP file format specification version 4.3, Jean-Loup Gailly and Mark Adler, 1996, http://tools.ietf.org/html/rfc1952
94. bzip2 and libbzip2, Julian Seward, 1996, http://www.bzip.org/
95. RFC 1122 Requirements for Internet Hosts - Communication Layers, R. Braden, 98
96. Community support for software development in small groups: the initial steps, Francesco Lelli and Mehdi Jazayeri, *SoSEA '09: Proceedings of the 2nd international workshop on Social software engineering and applications (978-1-60558-682-3)* p. 15-22, 2009, ACM Press, Amsterdam, The Netherlands, New York, NY, USA, http://doi.acm.org/10.1145/1595836.1595840
97. AT Internet Insitute, 2009, http://www.atinternet-institute.com/en-us/internet-users-equipment/operating-systems-february-2009/index-1-2-7-165.html
98. How package management changed everything, Ian Murdock, 2007, http://ianmurdock.com/solaris/how-package-management-changed-everything/
99. Filesystem Hierarchy Standard, FreeStandards.org, 2009, http://www.pathname.com/fhs/
100. Introduction to the File System Overview, Apple Inc., 2009, http://developer.apple.com/mac/library/documentation/MacOSX/Conceptual/BPFileSystem/BPFileSystem.html
101. The Old New Thing, Raymond Chen, 2007, http://blogs.msdn.com/oldnewthing/
102. Stop the madness: Subdirectories of My Documents, Raymond Chen, 2006, http://blogs.msdn.com/oldnewthing/archive/2006/12/28/1374334.aspx
103. RFC 1945, T. Berners-Lee, R. Fielding and H. Frystyk, 1996, http://www.ietf.org/rfc/rfc1945.txt
104. PHP, The PHP Group, 2009, http://www.php.net/manual/en/index.php
105. Streams API for PHP Extension Authors, The PHP Group, 2009, http://www.php.net/manual/en/internals2.ze1.streams.php
106. Programming Language Popularity, DedaSys LLC, 2011, http://langpop.com
107. At the forge: First steps with Django, Lerner, Reuven M. p. 11-, 2007, Seattle, WA, USA
108. Embedding Perl in HTML with Mason, Rolsky, Dave and Williams, Ken, *0596002254*, 2002, O'Reilly & Associates, Inc., Sebastopol, CA, USA
109. CPAN Frequently Asked Questions, Elaine Ashton and Jarkko Hietaniemi, http://cpan.org/misc/cpan-faq.html
110. Python Package Index, Python Software Foundation, 2010, http://pypi.python.org/pypi
111. Reporting guidelines for controlled experiments in software engineering, Jedlitschka A and Pfahl D p. 95-104, 2008, Empirical Softw. Engg.
112. Evaluating guidelines for reporting empirical software engineering studies, Kitchenham, Barbara, Al-Khilidar, Hiyam, Babar, Muhammed Ali, Berry, Mike, Cox, Karl, Keung, Jacky, Kurniawati, Felicia, Staples, Mark, Zhang, He and Zhu, Liming p. 97-121, 2008, Kluwer Academic Publishers, Hingham, MA, USA, 10.1007/s10664-007-9053-5
113. Guidelines for conducting and reporting case study research in software engineering, Runeson, Per and Host, Martin p. 131-164, 2009, Kluwer Academic Publishers, Hingham, MA, USA, 10.1007/s10664-008-9102-8
114. IPC::Run, Richard Soderberg, 2008, http://search.cpan.org/dist/IPC-Run/
115. LWP - The World-Wide Web library for Perl, Gisle Aas, 2009, http://search.cpan.org/~gaas/libwww-perl-5.831/lib/LWP.pm
116. WWW::Mechanize - Handy web browsing in a Perl object, Andy Lester, 2009, http://search.cpan.org/~petdance/WWW-Mechanize-1.60/lib/WWW/Mechanize.pm
117. LWP::Simple - simple procedural interface to LWP, Gisle Aas, 2009, http://search.cpan.org/~gaas/libwww-perl-5.831/lib/LWP/Simple.pm
118. Web client programming with Perl, Clinton Wong, 1997, O'Reilly & associates inc.
119. Kernel korner: intro to inotify, Love, Robert p. 8-, 2005, Seattle, WA, USA
120. Obtaining Directory Change Notifications (Windows), Microsoft, 2011
121. WxGlade - a GUI builder for WxWidgets, WxGlade Team, 2008, http://wxglade.sourceforge.net/index.php#description
122. Perl2Exe, IndigoStar Software, *IndigoStar*, 2007, http://www.indigostar.com/perl2exe.htm
123. PerlCC, Nicholas Clark, *CPAN - PerlCC*, 2007, http://search.cpan.org/~nwclark/perl-5.8.8/utils/perlcc.PL
124. Perl Archiving Toolkit, PAR Group, *Wikia - PAR*, 2007, http://par.perl.org/wiki/Main_Page
125. Perl Dev Kit 7, ActiveState, *ActiveState - Perl*, 2007, http://www.activestate.com/Products/perl_dev_kit/index.mhtml
126. LWP::Parallel::UserAgent - A class for parallel User Agents, Marc Langheinrich, 2009, http://search.cpan.org/~marclang/ParallelUserAgent-2.57/lib/LWP/Parallel/UserAgent.pm

127. perlref - Perl references and nested data structures documentation, Perl.org, 2011, http://perldoc.perl.org/perlref.html
128. PHP Language Data Types, The PHP Group, 2011, http://www.php.net/manual/en/language.types.intro.php
129. Charming Python: Numerical Python - Working with the numeric and numarray packages, David Mertz, 2011, http://www.ibm.com/developerworks/linux/library/l-cpnum/index.html
130. Perl FAQ part 6 - How can I hope to use regular expressions without creating illegible and unmaintainable code?, Perl.org, 2011, http://perldoc.perl.org/perlfaq6.html#How-can-I-hope-to-use-regular-expressions-without-creating-illegible-and-unmaintainable-code%3f
131. threads - Perl interpreter-based threads, Jerry D. Hedden, 2009, http://search.cpan.org/~jdhedden/threads-1.74/threads.pm
132. POE: Perl Object Environment, and , 2009, http://poe.perl.org/
133. Programming Perl, Larry Wall, *0596000278*, 2000, Sebastopol, CA, USA
134. wxPython, a GUI Toolkit, Hugues Talbot p. 5, Seattle, WA, USA
135. Writing PERL Modules for Cpan, Tregar, Sam, *159059018X*, 2002
136. Config::Abstract::Ini, Eddie Olsson, 2011, http://search.cpan.org/~avajadi/Config-Abstract-0.16/Ini/Ini.pm
137. Config::Any::INI, Brian Cassidy, 2011, http://search.cpan.org/~bricas/Config-Any-0.20/lib/Config/Any/INI.pm
138. AnyData::Format::Ini, Jeff Zucker, 2011, http://search.cpan.org/~jzucker/AnyData-0.10/AnyData/Format/Ini.pm
139. OpenPlugin::Config::Ini, Eric Andreychek, 2011, http://search.cpan.org/~eric/OpenPlugin-0.11/OpenPlugin/Config/Ini.pm
140. Config::Backend::INI, Hol Oester, 2011, http://search.cpan.org/~oesterhol/Config-Backend-INI-0.12/lib/Config/Backend/INI.pm
141. Config::IniHash, Jan Krynicky, 2011, http://search.cpan.org/~jenda/Config-IniHash-3.01.01/lib/Config/IniHash.pm
142. Config::INI::Access, Andrew Shitov, 2011, http://search.cpan.org/~andy/Config-INI-Access-0.9999/lib/Config/INI/Access.pm
143. Config::INI, Ricardo Signes, 2011, http://search.cpan.org/~rjbs/Config-INI-0.017/lib/Config/INI.pm
144. Config::IniFiles, Shlomi Fish, 2011, http://search.cpan.org/~shlomif/Config-IniFiles-2.66/lib/Config/IniFiles.pm
145. PEP 20 - The Zen of Python, Tim Peters, 2004, http://www.python.org/dev/peps/pep-0020/
146. Perl, the first postmodern computer language, Larry Wall, 1999, http://www.perl.com/pub/1999/03/pm.html
147. PyInstaller, William Caban, 2009, http://www.pyinstaller.org/
148. Py2Exe, Jimmy Retzlaff, 2009, http://www.py2exe.org/
149. Representation-based just-in-time specialization and the psyco prototype for python, Rigo, Armin, *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (1-58113-835-0)* p. 15-26, 2004, Verona, Italy, New York, NY, USA, http://doi.acm.org/10.1145/1014007.1014010
150. Markus F.X.J. Oberhumer, László Molnár and John F. Reiser, 2009, http://upx.sourceforge.net/
151. urllib2 - extensible library for opening URL, Python Software Foundation, 2009, http://docs.python.org/library/urllib2.html
152. urllib2 - The Missing Manual, Michael Foord, 2009, http://www.voidspace.org.uk/python/articles/urllib2.shtml
153. TRAMP: Makes RDF look like Python data structures, Aaron Swartz, 2008, http://www.aaronsw.com/2002/tramp/
154. XML::Simple, Grant McLean, 2008, http://search.cpan.org/dist/XML-Simple/
155. Pyinotify, Sébastien Martini, 2007, http://pyinotify.sourceforge.net
156. PyPy, and , 2009, http://codespeak.net/pypy/dist/pypy/doc/
157. Bringing dynamic languages to .NET with the DLR, Hugunin, Jim, *Proceedings of the 2007 symposium on Dynamic languages (978-1-59593-868-8)* p. 101-101, 2007, Montreal, Quebec, Canada, New York, NY, USA, http://doi.acm.org/10.1145/1297081.1297083
158. Programming Python, Part I, Jose P. E. Fernandez p. 2, 2007, Seattle, WA, USA
159. APT HOWTO, Gustavo Noronha Silva, 2005, http://www.debian.org/doc/manuals/apt-howto/
160. Seth Vidal, 2009, http://yum.baseurl.org/
161. Christoph Pfisterer, 2009, http://www.finkproject.org/
162. Google this!: using Google apps for collaboration and productivity, Herrick, Dan R., *SIGUCCS '09: Proceedings of the ACM SIGUCCS fall conference on User services conference (978-1-60558-477-5)* p. 55-64, 2009, St. Louis, Missouri, USA, New York, NY, USA, http://doi.acm.org/10.1145/1629501.1629513
163. Windows Live - Desktop and Online Services, Microsoft Corp., 2009, http://www.windowslive.com/Explore
164. Kernel Korner: A Non-Technical Look Inside the EXT2 File System, Appleton, Randy p. 19, Seattle, WA, USA
165. Advances in flash memory SSD technology for enterprise database applications, Lee, Sang-Won, Moon, Bongki and Park, Chanik, *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data (978-1-60558-551-2)* p. 863-870, 2009, Providence, Rhode Island, USA, New York, NY, USA, http://doi.acm.org/10.1145/1559845.1559937
166. Essential COM, Don Box, *0201634465*, 1997, Boston, MA, USA
167. SAX: A Simple API for XML, D. Megginson, http://www.saxproject.org
168. Expat - The XML Parser Toolkit, J. Clark., http://www.jclark.com/xml/expat.html
169. Windows Vista: implementation challenges, Gale Fritsche, *SIGUCCS '07: Proceedings of the 35th annual ACM SIGUCCS conference on User services (978-1-59593-634-9)* p. 113-117, 2007, Orlando, Florida, USA, New York, NY, USA, http://doi.acm.org/10.1145/1294046.1294072

170. IEEE POSIX Certification Authority, IEEE, 2003, http://standards.ieee.org/regauth/posix/
171. Winsock Reference (Windows), MSDN Developer Network, 2009, http://msdn.microsoft.com/en-us/library/ms741416%28VS.85%29.aspx
172. VB6Cli.exe Fixes License Problems with Visual Basic 6.0, MSDN Developer Network, 2009, http://support.microsoft.com/kb/194751
173. CSocketMaster 1.2 & CSocketPlus 1.1 - Winsock classes, Emiliano Scavuzzo, 2004, http://www.pscode.com/vb/scripts/ShowCode.asp?txtCodeId=54681&lngWId=1
174. The impact of software engineering research on modern progamming languages, Barbara G. Ryder, Mary Lou Soffa and Margaret Burnett p. 431-477, 2005, New York, NY, USA, http://doi.acm.org/10.1145/1101815.1101818
175. Binary compression rates for ASCII formats, Isenburg, Martin and Snoeyink, Jack, *Web3D '03: Proceedings of the eighth international conference on 3D Web technology (1-58113-644-7)* p. 173-ff, 2003, Saint Malo, France, New York, NY, USA, http://doi.acm.org/10.1145/636593.636619
176. Distributed COM: Application Development Using Visual Basic 6.0, Maloney, Jim, *0130213438*, 1999, Upper Saddle River, NJ, USA
177. Visual Basic, Erik Meijer, *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion (978-1-59593-865-7)* p. 860-861, 2007, Montreal, Quebec, Canada, New York, NY, USA, http://doi.acm.org/10.1145/1297846.1297926
178. What's New in Windows Forms 2.0 (Level 200), Microsoft Corporation, http://msevents.microsoft.com/CUI/WebCastEventDetails.aspx?EventID=1032271754&EventCategory=3&culture=en-US&CountryCode=US
179. Error Handling, ESRI Developer Network, 2005, http://edndoc.esri.com/arcobjects/8.3/GettingStarted/ErrorHandling.htm